

送付先：本部長・部長・主任技師（人材）

2015年度 海外業務研修報告書

オープンソースソフトウェアの機能安全対応に関する研究

所属 : 組込みエンジニアリング事業部
組込みシステム本部 第二設計部 第一グループ

氏名 : 橋本 耕太郎

派遣先 : Hitachi India Pvt. Ltd. Bengaluru R&D Centre

派遣期間 : 2015年10月3日～2016年3月28日

内容

1. 研修の目的・目標
2. 業務研修内容
3. 習得事項
4. 今後の業務への貢献・活用
5. 現地の生活や文化の紹介

2016年4月18日

概要

近年、自動車や産業・インフラ機器の制御に Linux に代表されるオープンソースソフトウェア (OSS) が広く利用される動向があり、特に安全性が厳しく要求される用途に対して OSS で高いレベルの品質を実現する必要性が高まっている。しかし一般に OSS 自体には機能が正しく動作することの保証がない。そのため OSS で人命が関わるようなシステムを構築する場合は、OSS を利用する開発者がシステムが正しく動作することの証明を与えるか、または不足している品質を補完する必要がある。

SIL2LinuxMP は OSS・Linux システムで一定の機能安全水準を達成するための方法論確立を目的とするプロジェクトで、日立は 2015 年 4 月のプロジェクト立上げ時からレビューパートナーとして参画している。SIL2LinuxMP プロジェクトでは主に機能安全規格 IEC61508 で定められている安全要件を OSS で実施するためのプロセス策定と、安全要件を満たすことを示すためのソフトウェア検証技法の調査・開発が実施されている。本報告書は筆者が 2015 年 10 月からの半年間 Hitachi India Laboratory で実施した SIL2LinuxMP プロジェクト活動の内容とそこで得たノウハウを次の事項ごとにまとめたものである。

1. OSS・Linux に対するソフトウェア検証手法・ツールの調査

OSS の機能安全対応に必要な技術要素として、バグ検出、テストケース生成、反例検出、メトリクス測定、False-Positive 抑制をサポートする技法の調査を行った。OSS ソフトウェアの検証としては、fuzzer としての性質を持つツールでカバレッジ向上を図るなど、様々な検証作業を自動化する戦略が有効となる場合が多いことが分かった。

2. SIL2LinuxMP の戦略と規格準拠プロセスの理解

各検証技術をどのように組み合わせて利用すれば IEC61508 の安全要件を満たす結果を得ることができるかを検討した。OSS の機能安全対応を実現するには検証対象と開発ツールを最小構成とすることが有効である。また、関数コールグラフを応用することで、テストカバレッジの測定やアプリケーション同士の独立性解析が実施できることが分かった。

3. 質疑や議論を通した SIL2LinuxMP コミュニティへのコントリビューション

メーリングリストと Git で SIL2LinuxMP コミュニティに対して質疑や技術提案を行うことを継続した結果、パートナーのうち最も活発に活動した履歴を残すことができ、一連の日立の活動がコミュニティの活動推進に役立っているとの評価を受けた。

4. CodeMinimization 技法の開発と応用例提案

コンパイル対象とならないコードを削除することで検査対象を限定する技法の開発を行った。本手法はソースコードの可読性を向上させ、検証コストを抑制し、カバレッジメトリクスの正確性を高める効果があり、さらにあらゆる検証技術のパフォーマンスを向上させる可能性を持つ。

目次

概要	1
第 1 章 研修の目的・目標	1
1.1 OSS・Linux を使ったシステムの機能安全認証を実現する方法の研究	1
1.2 ソフトウェア形式検証や機能安全の専門知識を持つ海外のエンジニアとの関係構築	2
第 2 章 業務研修内容	3
2.1 OSS・Linux に対するソフトウェア検証手法・ツールの調査	3
2.2 SIL2LinuxMP の戦略と規格準拠プロセスの理解	4
2.3 質疑や議論を通した SIL2LinuxMP コミュニティへのコントリビューション	4
2.4 Code Minimization 技法の開発と応用例提案	5
第 3 章 習得事項	6
3.1 OSS・Linux に対するソフトウェア検証手法・ツールの調査	6
3.1.1 バグ検出	7
Coccinelle	7
Csmith	10
3.1.2 テストケース生成	12
KLEE	12
3.1.3 反例検出	13
CPAChecker	13

3.1.4 メトリクス測定	16
CodeViz	16
syzkaller	18
3.1.5 バグ選別	19
Herodotos	19
Prequel	20
bugspots	21
3.1.6 プロジェクト管理	22
OGSN	22
rmToo	24
3.1.7 その他	25
3.2 SIL2LinuxMP の戦略と規格準拠プロセスの理解	26
3.2.1 SIL2LinuxMP のツール選定戦略	26
スケールする検証フレームワークを実現するための戦略	27
3.2.2 統合検証フレームワーク：DB4SIL2	28
関数コールグラフとトレーサを用いたカバレッジ解析手法	30
関数コールグラフの LOPA への応用	32
seccomp と cgroups のリソース独立性を解析する例	33
seccomp と cgroups の Common Cause Failure (CCF) 分析例	34
3.3 質疑や議論を通した SIL2LinuxMP コミュニティへのコントリビューション	36
3.3.1 メーリングリスト上での活動	36
3.3.2 Git 上での活動	40
SIL2LinuxMP が開発するツールへのフィードバック	41
検証ツール調査内容の共有	41
日立が独自に開発した検証手法やツールの公開	41

	SIL2LinuxMP Git レポジトリでの活動結果	42
3.3.3	Face to face meeting	43
3.4	Code Minimization 技法の開発と応用例提案	45
3.4.1	問題設定	45
3.4.2	実現方針	47
(i)	既存 Makefile の拡張	47
(ii)	プリプロセスコマンドの構築と実行	48
(iii)	展開されたヘッダファイルの中身と余分な空白行の削除	48
3.4.3	実行結果	49
3.4.4	Code Minimization 技法の利点と応用	51
	変換後のソースコード複雑度評価	51
	検査対象範囲の最小化	55
	関数コールグラフを枝刈りする応用例	55
	部分ソースツリーの抽出	57
3.4.5	まとめと今後の課題	58
第 4 章	今後の業務への貢献・活用	59
4.1	OSS・Linux システムの機能安全対応パイロット開発実施	59
4.2	機能安全対応 Linux ディストリビューションの構築と提案	60
4.3	SIL2LinuxMP コミュニティへの継続的なコントリビューション	62
第 5 章	現地の生活や文化の紹介	63
5.1	平日の様子	63
5.2	週末の様子	64
5.3	交通事情	65
5.3.1	バス	65

5.3.2 鉄道	67
5.3.3 メトロ	69
5.3.4 リキシャ	70
5.3.5 タクシー	70
5.3.6 徒歩	71
5.4 旅行	71
5.4.1 ムンバイ、オーランガバード、エローラ・アジャンタ	71
5.4.2 マイソール	73
5.4.3 ハンピ	74
5.4.4 スリランカ	75
5.4.5 ドイツ	78
5.5 食事	79
5.6 英語	80
謝辞	81
参考文献	82

第1章

研修の目的・目標

1.1 OSS・Linuxを使ったシステムの機能安全認証を実現する方法の研究

近年、組込みシステムから業務ソフトまであらゆる IT ソリューションに対する要求が大規模化・複雑化しており、従来は各社が独自にクローズドに開発していた領域でもオープンソースの力を利用しないと競争力のあるソリューション開発や提案が現実的にできなくなりつつある。世界的に IT 業界がオープンソースへシフトしていく流れの中で、自動車、鉄道、産業機器分野など安全性が厳しく要求される分野でも同様に GNU/Linux をはじめとしたオープンソースを利用した開発が行われている。このような人命が関わるようなシステム開発は国際的に定められた機能安全規格 (IEC61508, ISO26262 など) に準拠し第三者機関によって認証を取得することが事実上必須の市場参入要件となりつつある。これまで、機能安全対応の開発を行う場合は開発対象となる全ての部品 (ハードウェア・ソフトウェア) について開発プロセスを厳密に定め全てを自社内で管理・評価することが一般的であったが、オープンソースソフトウェアを利用する場合は次の 2 点の前提が問題となる。

- オープンソースソフトウェアそれ自体は安全性を保証しない
- コミュニティごとに独自流儀の開発方法を持っている

このような特性を持つ第三者が開発したオープンソースソフトウェアを部品として統合し機能安全対応の製品を構成するためには、利用する部品のどの箇所が規格に準拠していないかを特定して、規格に準拠しない箇所を補完するような対策を講じる必要がある。このための、OSS・Linux を使用したシステムの機能安全対応プロセスおよびソフトウェア検証手法の確立を目的としたプロジェクト SIL2LinuxMP が 2015 年 4 月に発足した。日立は SIL2LinuxMP プロジェクト開始当初からレビューパートナーとして参加しており、SIL2LinuxMP コミュニティと連携した活動を Hitachi India Laboratory (HIL) が中心となって進めている。HiICS は OSS・Linux システムの機能安全対応または一般に品質保証の分野で自社のビジネス拡大を見据えており、日立の SIL2LinuxMP への活動をスポンサーしている関係にある。本研修の第一の目標は、HIL の研究者と共同で OSS・Linux システムの機能安全認証プロセスとソフトウェア検証技法の調査研究を行うことで、HiICS のビジネスに還元できるようなノウハウを得ることである。

1.2 ソフトウェア形式検証や機能安全の専門知識を持つ海外のエンジニアとの関係構築

SIL2LinuxMP プロジェクトではリファレンスとなる製品とユースケースを想定して機能安全認証プロセスの策定および検証手法の開発を行う。しかし実際に市場に出る製品の機能安全対応を行う際は、SIL2LinuxMP のリファレンスで確立されたプロセスと手法をそのまま再利用することはできず、必ず個々の製品仕様とユースケースに応じてカスタマイズしたプロセスと手法が必要となる。そのためには機能安全規格書を読み解き、独自の製品に対応して解釈・対応することが必要になるが、一般にソフトウェア技術者にとって IEC61508 等の規格書は難解であるため認証規格を熟知したエキスパートの協力が必要不可欠となる。SIL2LinuxMP プロジェクトは Linux Kernel の産業用途向けリアルタイム対応を推進するドイツの非営利団体 Open Source Automation Development Lab (OSADL) と OpenTech が主催している。プロセス策定に関わるパートナーとして BMW, Intel, Bosch, Elektrobit, KUKA, Renesas 等の産業機器向けアプリケーションを持つ企業が、技術支援を行う学術団体としてフランスの研究機関 INRIA およびロシアと中国の研究機関が、機能安全認証機関として TUeV Rheinland が参画している。本研修の第二の目標は、日立から SIL2LinuxMP コミュニティに対して認証プロセスや検証手法のレビュー、質疑応答、技術的フィードバックを行うことを通して、プロジェクトに参加している各分野のエキスパートと意見を交換できるような関係を構築していくことである。これにより SIL2LinuxMP プロジェクト終了後も機能安全やソフトウェア検証に関する知見を継続的に蓄積できるようにし、また日立のオープンソースコミュニティへの活動と貢献を外部にアピールしていくことが目的である。

第2章

業務研修内容

2.1 OSS・Linuxに対するソフトウェア検証手法・ツールの調査

オープンソースソフトウェア自身で保証されていない品質を補完するためには各種ツールの利用が有効である。ツールは認証対象の補完すべき品質に対するエビデンスを生成するという目的に従って選定・評価して用いる。品質保証のためのツールとして一般的なものにはソースコードの静的検証やテスト自動化を目的とする検証ツールがあるが、それ以外にも以下のような様々なタスクが品質保証には求められる。

- テストケース自動生成
- 動的回帰テスト
- バグ抽出 (False-Positive フィルタリング)
- バグレポート生成・バグ管理
- テスト結果蓄積
- メトリクス測定・テストレポート生成

特にテストケース・テスト結果・バグのマネジメントは重要であるが、人手で行うには非常に手間がかかるタスクである。ゆえにこれらのタスクからなる品質管理ライフサイクルをいかにして効率的に回すことができるかが持続的な品質管理にとって決定的に重要である。これらのタスクをサポートまたは自動化するツールはオープンソースライセンスで利用可能なものが多数存在する。ただしプロジェクトによってはツール自体が十分に成熟していなかったり、開発が中断していたり、目的に沿った結果が得られない等の場合がある。そのため、予めどのような用途にはどのツールが利用可能でどのプロジェクトでどれだけの利用実績があるかを調査すること、および自身の手で実験・評価をすることが重要である。また、SIL2LinuxMP プロジェクト自体でもいくつか特定のツールが採用候補として検討されているため、それらを独自に評価して用途、使用方法、解析方法、拡張性、制約、性能を理解しておくことが必要である。本研修では、機能安全または品質保証に適用できる可能性のあるツールおよび技法に対する調査を行った。

2.2 SIL2LinuxMP の戦略と規格準拠プロセスの理解

SIL2LinuxMP では、鉄道、自動車、航空機、産業機器、医療機器など各分野に特化した機能安全規格ではなく、それらを広くカバーする”umbrella standard”である IEC61508 を対象とする。現状 OSS・Linux システムを対象とした機能安全規格は存在せず、OSS・Linux システム一般に機能安全認証方法論の確立を正攻法で目指す試みとしては SIL2LinuxMP が初めてである。SIL2LinuxMP のリファレンスユースケースを対象とした認証プロセスを確立できた後は、それがモデルケースとして再利用され各ベンダが各自のアプリケーションに適用・拡張することが想定されている。

機能安全規格 IEC61508 には製品の認証方法として以下の 3 通りの方法が示されている (IEC61508-3 7.4.2.12)。

- Route 1_S: compliant development (初めから最後まで全て厳格に品質管理されたプロセスの下で開発・評価を行う)
- Route 2_S: proven in use (十分な使用実績をエビデンスとして認証の根拠とする)
- Route 3_S: assessment of non-compliant development (規格非適合箇所に対して対策を行い適合と証明するに十分な根拠を与える)

SIL2LinuxMP ではこのうち Route 3_S: assessment of non-compliant development を採用する。オープンソースソフトウェアは第三者が各々独自のスタイルで開発しているため 1_S のように厳格な品質管理がされているという保証が一般に得られない。また 2_S で認証できるものは特定のハードウェアとソフトウェアおよびその構成と固定された設定のみであり、これは将来 SIL2LinuxMP の成果を再利用して多様なアプリケーションに拡張する方針であることに反する。SIL2LinuxMP の Route 3_S による基本的なアプローチは、製品の部品となるソフトウェアおよび開発ツールについて採用基準となる評価項目を IEC61508 に従って定め、候補となるオープンソースのリソースから適切なものを選定・評価することである。3_S による方法では、定義された選定・評価により開発・検証ツールセットを構成して規格非適合箇所を補完するエビデンスを生成し、それをもって機能安全認証機関 TUeV Rheinland に説明を行うこととなる。本研修では、選定された検証ツールの使用方法、組み合わせ方、および認証機関 TUeV Rheinland を納得させるためのエビデンス生成方法に焦点を当てて調査研究を行った。

2.3 質疑や議論を通した SIL2LinuxMP コミュニティへのコントリビューション

SIL2LinuxMP コミュニティに対して活動をする意図の一つは HiICS がビジネスに転換するためのノウハウを引き出すことであるが、それに加えて、日立からの貢献を OSS コミュニティにアピールするという目的がある。

オープンソースが産業界でまだメジャーではなかったころは企業が独自技術で顧客を広く囲い込むこと

で利益を上げる戦略が有効であり、独自技術を OSS 化して一般に公開することは利敵行為であると捉えられていた。むしろ、OSS 資産はフリーライドして自社ビジネスに利用するべきものという理解が一般的であった。しかし近年、様々な技術を OSS 化すること、また OSS プロジェクトに貢献することへの価値あるいはビジネスモデルの転換が起きている。LinuxCon のような技術カンファレンスでは、キーノートや個々の発表で「何々の OSS プロジェクトにコントリビュートしている企業 TOP10」のような紹介が頻繁に行われている。そこでは個人名ではなく企業名ごとの貢献度ランキングが明らかになり、貢献度の大きい企業は技術者からのリスペクトの対象となる。またその事実がその企業のサービスやビジネスを裏付け、さらには第三者が活躍できる場を提供できる技術力を持つことが広告材料となり他社と差別化されることにつながっている。それはまさに世界中の大手 IT 企業がメジャーな OSS プロジェクトの陣取り合戦をしている状態であり、その本流に一つも貢献の実績がなくフリーライドばかりしているような企業は非常に弱い立場にあるのが実情である。このように、自社の活動をオープンな場でアピールし、具体的な実績として OSS プロジェクトへの貢献を行うことで技術市場の占有率を高めることは投資的な意味で非常に重要である。

参考ブログエントリ：[The software, be open or die \[17\]](#)

企業の OSS 貢献度としてオープンに評価されるほぼ唯一の指標が本流（アップストリーム）への修正・追加コード適用数である。筆者が以前 Android プラットフォーム開発に携わっていたとき、そこでは製品となるソースコードに独自のカスタマイズや修正を大量に加えるスタイルで開発が行われていた。そこに元のソースコード（アップストリーム）のアップデートが行われると、現状のソースコードとの差分解釈およびカスタマイズ部分の抽出をした上で新しいソースコードにカスタマイズ部分を加え直すという作業が発生し、これがしばしばプロジェクトが混乱する原因となっていた。もし我々がアップストリーム側の開発に関わることが出来ていれば、このような製品開発にとって本質的でない作業をいくらか緩和する対策を講じることができた可能性がある。

SIL2LinuxMP の場合はメインに開発する対象がソースコードではなくドキュメントであるという違いはあるものの、プロジェクト管理は Git とメーリングリストが用いられており各参加者の活動は記録され統計情報として定期的に公開される。OSS 活動は必然的に社外の技術者との協調が求められ、特に何らかのアクションを起こす際には「何のためになぜそれが必要か」ということを十分に説明しコミュニティに納得してもらうというハードルが存在する。筆者はこの機会を OSS 活動に必要なスキルを身につける場と位置づけ、多くのかつ意味のあるメーリングリスト投稿と Git コミットを行うことを課題とした。

2.4 Code Minimization 技法の開発と応用例提案

筆者が HIL での業務研修を開始する 2015 年 10 月より以前に、日立はソフトウェア検証手法のパフォーマンスを向上させる技法 Code Minimization を開発し SIL2LinuxMP コミュニティに対して提案をしていた。しかしその技法を実現するための実装が不完全で使用方法や出力が洗練されておらず、さらにその効果を実証するための例を上手く挙げることができていなかったため、コミュニティからの反応がほとんどない状態であった。筆者はこの技法について実装とドキュメントを一般に公開できるレベルにまで洗練し、かつ SIL2LinuxMP コミュニティが興味を示すような具体的応用例を示すことでこの技法の効果への理解を得ることを課題とした。

第3章

習得事項

3.1 OSS・Linuxに対するソフトウェア検証手法・ツールの調査

本節では、OSS・Linux システムのソフトウェア検証に関する技法またはツールについて、SIL2LinuxMP プロジェクトで採用が検討されているものを中心に調査結果を記載する。前述のように、品質管理ライフサイクルにおいてはテスト実行のみにとどまらずテストケース生成、バグ抽出（False-Positive フィルタリング）、バグレポート生成、バグ管理、メトリクス測定・テストレポート生成など様々なタスクを実施する必要がある（図 3.1）。

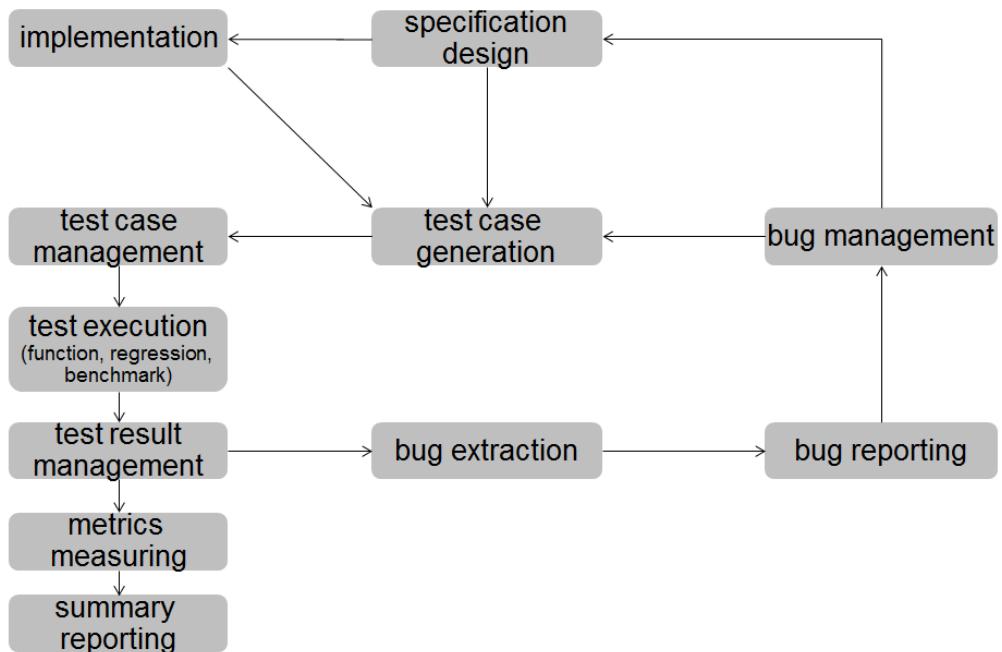


図 3.1 品質管理ライフサイクルを構成する様々なタスク

これらのタスクをサポートするツール類としては、図 3.2 に示すオープンソースの資産が利用可能である。ただし図 3.2 に示したツール類はあくまでも一例であり、これらのツールを利用すれば各々のタスク

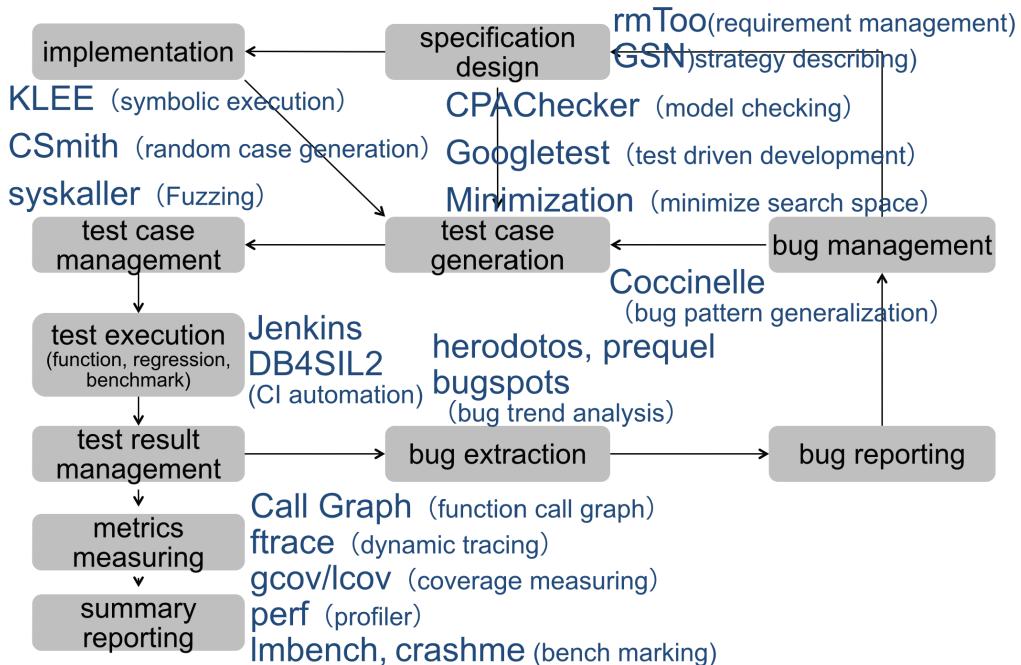


図 3.2 品質管理ライフサイクルの各タスクで利用できるツール類の例

を完璧に実施できるという万能なものでもない。ツールはそれぞれ適用可能なソフトウェアターゲット、得意/不得意な実施内容、実行環境上の制約など様々な性質を持つ。ツールはあくまでも手段に過ぎず、実際に品質管理ライフサイクルを考える際は「何に対して何を行いたいのか」という目的に対して必要な基準を満たすツール・手法を選定するというアプローチをとることが原則である。以降は図 3.2 中のツール・手法のうち目的別にいくつかを取り上げそれぞれの調査結果を述べる。

3.1.1 バグ検出

Coccinelle

Coccinelle [7] は C ソースコードのパターンマッチング・変換エンジンである。例えば、ある API を使用している箇所全てについて引数を拡張することや別の API に置換するという処理を一括して行うこと、またある論理式・計算式について誤っている表記を一斉に修正すること、さらに C 言語の表記を統一するなどコードリファクタリングのような処理を一括して行うことが可能である。ソースコード中の特定のパターン検索には Semantic Patch という非常に柔軟で強力な表現が可能な言語が用いられる。Semantic Patch はスペースや括弧などの表現のばらつきや、変数・関数名の相違などを柔軟に吸収することができ、式の形やデータ構造、処理の流れといったプログラムの意味的な単位に対して作用する。Linux 環境におけるパターンマッチングの技術としては正規表現を使った sed コマンドが使われることが多いが、Semantic Patch を使った Coccinelle によるパターンマッチングはそれよりも強力であるという意味で”doped sed”と表現されることがある。Coccinelle は”Find once, Fix anywhere”的思想のもとで開発が進められている。これには、一度 Semantic Patch でバグパターンを表現してしまえば同種の類似バグを全て改修できる仕組みを作ることで、何度も至る所で同じ修正を繰り返すという無駄な作業を撲滅するという意図が込められている。

Coccinelle はフランスの研究機関 INRIA が開発・保守している。Coccinelle は関数型言語の OCaml で書かれており、OCaml 自体も INRIA で開発されている。Coccinelle は Linux Kernel のバグ修正に 1200 件以上の適用実績がある。Coccinelle の Linux Kernel におけるワークフローは、新たに見つかったバグを一般化する形で Semantic Patch に書き直して再びソースツリー全体に適用することでその Semantic Patch パターンにマッチする同種の残存バグを抽出するというものである。以下に、実際に Linux Kernel に適用実績のある論理演算誤りを修正する Semantic Patch の例を示す。

C 言語において not 演算子”!”は and 演算子”&”よりも先に評価されるが、この優先順位関係を誤って認識していたために意図とは異なる論理式を書いてしまう誤りパターンが存在する。

not 演算子と and 演算子の優先順位を誤った論理演算を含むコード例

```
if(!dma_cntrl & DMA_START_BIT) {  
    BCMLOG(BCMLOG_DBG, "Already Stopped\n");  
    return BC_STS_SUCCESS;  
}
```

この誤りに対して、一般に”!式 & 定数”のパターンに当たるコードを”!(式 & 定数)”の表記に一括修正する Semantic Patch は以下のようになる。

”!式 & 定数”のパターンを”!(式 & 定数)”の表記に一括修正する Semantic Patch

```
@@  
expression E;  
constant C;  
@@  
  
- !E & C  
+ !(E & C)
```

ここで、@@で囲まれた箇所はメタ変数宣言部であり、パターンサーチする対象として式 (expression)、文 (statement)、型 (type)、定数 (constant)、ローカル/グローバル変数 (idexpression) などを指定することができる。ここで宣言したメタ変数を使用してパターンの変換ルールを Semantic Patch 文法に従って表現していく。この Semantic Patch を上記誤りを含むソースファイルに適用すると、Coccinelle はパターン変換結果を具体的な diff 形式で出力する。

Coccinelle によって検出されたパターン変換結果

```
- if(!dma_cntrl & DMA_START_BIT) {  
+ if(!(dma_cntrl & DMA_START_BIT)) {
```

Coccinelle は常にパターンの変換のみを行うものではなく、Semantic Patch の書き方によっては Coccinelle の出力を diff 形式ではなくパターンマッチしたファイル名と行数のみの表示とするという制御も可能である。上記のようにして Coccinelle の出力が得られた後は、それら Coccinelle の指摘が確かに

に真のバグであってソースコードに修正を適用するべきものであるかを開発者自身が確認することになる。ここで Semantic Patch の書き方が洗練されていないと、修正する必要のないコードに対しても無駄に Coccinelle がパターンマッチしてしまい、False-Positive が大量に出力されてしまうことがある。

これまで実際の Linux Kernel 開発でパターン化されバグ修正に貢献実績を持つ Semantic Patch が [Coccinellery: A gallery of semantic patches \[8\]](#) で蓄積・公開されている。Coccinellery には、式のマクロへの置き換え、API 引数追加・削除、API 置換、同値表現の統一、デッドロック検出、メモリリーク検出、無効な論理演算検出など Linux Kernel 開発で過去にあった既存の修正パターンがほぼ網羅されている。これらの Semantic Patch は再利用することが可能であるが単に使い回せばバグが簡単に検出できるわけではなく、適用する Semantic Patch の意図と使い方および期待する出力を各々把握することが必須である。Semantic Patch の内容を把握しないまま単に Coccinelle を実行してもそれらの結果を解釈することができず、また真の指摘が大量の False-Positive に埋もれてしまうこととなる。Coccinelle の False-Positive に対しては、[Semantic Patch の文法 \[27\]](#) に基づいてパターンマッチがより精度よくかつ出力が分かりやすい形となるよう Semantic Patch を推敲することが根本的な対策となる。Coccinellery に蓄積されている Semantic Patch は既に Linux Kernel に適用済みのものであるので、それらを全て再度適用することにはあまり意味がない。蓄積された Semantic Patch の一つの使い方としては、ソースコードに変更を加える前後で Coccinelle を実行して得られた指摘を False-Positive 含めて比較し、変更前に無く変更後にのみ存在する差分指摘を得ることで回帰テストを効率化する戦略が考えられる。

Coccinelle は汎用的なセマンティックパターンサーチエンジンであるため、ソースコードのバグ検出以外にも広く様々な使い道が考えられる。例えばセキュリティ対策としてのコード難読化 (code obfuscation) のために、以下のように意味的に等価なコード変換が Coccinelle で実現できる可能性がある。

- 関数名、変数名のランダム文字列への変換
- 数値演算、論理演算の等価で複雑な形への変換
- 制御構造の複雑化（例：ループ制御の再帰呼び出しへの等価変換）
- 関数呼び出し関係の値渡しから参照渡しへ等価変換
- データ構造の複雑化（例：構造体の分解または統合）

また、Coccinelle の派生ツールとして、[Coccigrep \[6\]](#) というツールがある。Coccigrep は Semantic Patch の文法に基づいて C ソースコード中でセマンティック検索するツールで、次のような解析手段を提供する。

- 特定の型または特定の構造体の変数がどこで使われているか調べる
- 特定の構造体のフィールドがどこで参照されているか調べる

Csmith

Csmith [14] は C プログラムのランダムジェネレータであり、コンパイラの動作検証を行う目的で使用される。Csmith は GCC、LLVM、および商用コンパイラの自動検証とバグ検出に長い実績がある。特に、コンパイル最適化オプション有効時の誤り検出に顕著な貢献をしてきた。

機能安全対応のソフトウェア開発においては、開発ツール自体についても厳格に採用基準を定め評価を行なうことが求められる。IEC61508-4 3.2.11において開発ツールは認証対象となるソフトウェアへの影響度に応じて T1, T2, T3 の 3 段階のクラス分けがなされており、そのうちコンパイラはソフトウェア（バイナリ）生成に直接関与するツールとして T3 クラスに分類される。コンパイラの誤動作は直ちにソフトウェア誤動作の要因となるため、コンパイラの正当性評価には膨大な量のテスト結果と使用実績がエビデンスとして求められる。実績として十分なエビデンスを得るためにには認証対象となるコンパイラについて固定されたバージョンとコンパイルオプションの下、様々な用途の数多くのアプリケーション適用例が必要である (IEC61508-3 7.4.4.2)。そのためアプリケーション開発元が単独でコンパイラ認証を行うことは事実上困難であり、機能安全対応のソフトウェア開発には第三者機関によって認証取得済のコンパイラを利用するかまたは専門のコンパイラ評価サービスを利用することが一般的であった。しかし SIL2LinuxMP が対象とするソフトウェアはオープンソースであり、開発ツールも全てオープンソースでなければないという制約があるため、次の 2 つの理由からコンパイラ検証を第三者に依存することができない。

- 認証対象プラットフォームを構成する OSS ソフトウェアのアップデートに依存してコンパイラも持続的なアップデートを前提とするためバージョンを固定できない
- コンパイラの検証手法自体もオープンな技術である必要があり、特定の第三者の技術に vendor lock-in される状態は許容されない

以上の制約から、機能安全対応の OSS・Linux プラットフォーム開発で使うコンパイラ検証プロセスは、オープンな技術であり、アップデートにも耐えられるよう検証プロセスが自動化可能で、過去の検証資産を再利用して反復適用できるような性質を持つものである必要がある。Csmith で生成したエビデンス自体ではコンパイラの正当性を証明するのに十分ではないものの、Csmith は SIL2LinuxMP プロジェクトで求められる必要な性質をもつコンパイラ検証ツールとして有効である。

Csmith は次のようなワークフローでコンパイラの正当性検証を行う。ここでは例として gcc-4.9 を最適化オプション -O2 で使用するときの検証方法を挙げる。

1. csmith コマンドで C プログラムをランダム生成する。

```
$ csmith -s <seed> random.c
```

2. 生成したプログラムを検証対象を含む複数のコンパイラにかける

```
$ gcc-4.8 -O0 random.c -o random_4.8_00
```

```
$ gcc-4.9 -O0 random.c -o random_4.9_00
```

```
$ gcc-4.9 -O2 random.c -o random_4.9_02
```

3. 生成した複数のバイナリを各々実行し出力結果を比較する

```
$ ./random_4.8_00
checksum = 54A570D1 … gcc-4.8 の-00 でコンパイルしたバイナリの実行結果
$ ./random_4.9_00
checksum = 54A570D1 … gcc-4.9 の-00 でコンパイルしたバイナリの実行結果
$ ./random_4.9_02
checksum = 54A570D1 … gcc-4.9 の-02 でコンパイルしたバイナリの実行結果
```

全ての実行結果の出力 (checksum 値) が一致すれば、ここでランダムに生成したプログラム random.c に関する検証は成功である。もしいずれかの結果が他と異なる出力となる、またはバイナリ実行が失敗する、またはコンパイル自体が失敗する場合は、その事象を該当コンパイラおよびオプションの組合せに関するバグとして調査する。以上の手続きを、csmith で生成するランダム C プログラムの種を変えながら繰り返し実施し続ける。検証成功したランダムプログラム数が増えるほど、その実績がそのコンパイラとオプションの組み合わせに対する品質を保証するエビデンスとなる。

Csmith が生成するランダムプログラムがコンパイラの正当性を示すために十分に有効なテストケースとなるためには、そのプログラムが C 言語で表現可能な出来る限り多様なコードを含んでいかなければならない。しかしこの要件は、コンパイラのテストケースとして使われるプログラムは C 言語仕様で不定または未定義の振る舞いをするコードを含んでいてはいけないという制約とトレードオフの関係にある。例えば、 $a = f(b) + g(b)$ という式について $f(b)$ と $g(b)$ のどちらが先に評価されるかは不定である。 $f(b)$ と $g(b)$ が共に b を変更するとき、コンパイラの実装によっては $f(b)$ と $g(b)$ の評価順序が逆転してその結果式全体の評価結果 a が異なる可能性があるが、この挙動は C 言語仕様では不定であるのでどちらの挙動も正しい。また、ゼロ除算、-1 での剰余、配列領域外アクセス、NULL ポインタ開放、整数オーバーフロー、未初期化変数の参照など C 言語仕様上未定義の動作となるコードは、コンパイラによっては気を利かせた挙動をするものがあるものの、ほとんどの場合はプログラムがクラッシュする結果となるため有効なテストケースとならない。このように、コンパイラの正当性検証テストケースとしては C 言語仕様で厳密に定義された振る舞いをするコードのみからなるプログラムを使用しないと、上記の Csmith のワークフローでは False-Positive が大量に生成されてしまう。

このため、Csmith のアルゴリズム開発では当初から現在まで「不定または未定義挙動のコードを含まずコンパイルエラーにならないコードで可能な表現を多く含むランダムプログラム生成」をいかに正確に実現するかに努力が払われてきた。生成される False-Positive には不定/未定義挙動が原因であるものと既知問題であるものがあり、真のバグ選別および有効なバグレポートの作成がとても困難であるという経験から、Csmith を使ったコンパイラ検証プロセスの自動化とバグトリアージ (False-Positive の山から真の指摘を抽出する) 方法の開発が課題であるとされている。これまで Csmith が適用された実績がある GCC はごく一部のバージョンとオプションの組合せに対してのみであり、GCC のアップデートに対して最新のバージョンと最適化オプションの組合せ検証網羅および Csmith による検証作業が追いついていない状況である。また Csmith による検証自体をどこまでやれば十分であるかを一様に判断することが困難である。機能安全対応の際は、例えば蓄積されたテスト結果を用いて「統計的に残存バグがあと何件以下」というような論理的な根拠を構築して認証機関に十分性を示すことが求められる。

3.1.2 テストケース生成

KLEE

KLEE [22] は Symbolic Execution ツールの一種で、既存のソースコードを元に同値分割テストケースを自動生成する目的で使用できる。KLEE は与えられた C ソースコードから可能な実行パスに遷移するための引数や変数の状態組合せを探索して模擬実行する (Symbolic Execution)。以下に KLEE が C ソースコードを元にテストケースを生成する挙動の例を示す。

テスト対象関数の例

```
int get_sign(int x) {
    if(x == 0)
        return 0;

    if(x < 0)
        return -1;
    else
        return 1;
}
```

上記関数に対して KLEE は以下のように振る舞う。

1. 関数中の全分岐 (if-else) を特定する
2. 特定した各分岐に遷移するための変数 x の条件を探索する
3. 探索した各条件について具体的な代表値を一組決める

この結果 3 パターンの $x = \{0, 16843009(> 0), -2147483648(< 0)\}$ で `get_sign()` を呼ぶテストケースが生成される。生成された 3 つのテストケースはそのまま何度も再利用できるため例えば回帰テストへの応用が考えられる。

IEC61508 は同値分割テストケース生成について仕様に基づく方法とプログラムの内部構造に基づく方法の 2 通りを示唆しており (IEC61508-7 Ed2 C.5.7)、このうち KLEE はプログラムの内部構造からテストケースを生成するツールとして適用できる。KLEE ではソースコードがあればテスト対象の仕様を特に知らなくてもテストケース生成が可能であるが、関数引数をシンボリック変数にしてテスト対象を呼び出すためのテストドライバの作成は必要となる。

KLEE のアルゴリズムはヒューリスティック探索に基づくため 100% 分岐網羅を保証するものではない。上記の例のように単純なプログラムであれば 100% 分岐網羅を達成することができるが、実用上の大規模なプログラムでは計算コスト上の制約が問題となる。KLEE のパフォーマンスを向上させるためには、例えば無駄な探索をさせないようにシンボリック変数の動く範囲を指定することが有効である。テスト対

象となる関数の引数が数値ならばその上限と下限を、文字列ならば各文字が取り得る値を、または論理式を使い複数の引数の制約関係を表現することもできる。これを利用すれば、シンボリック変数の条件をテストドライバ側で指定することで入力仕様に基づいたテストケースを設計することができる。また、用意された4種類の探索アルゴリズムから適当なものを選択したり、アルゴリズムをラウンドロビンで組合せたり、最大探索深さやタイムアウト時間の設定ができたり、優先して探索するシンボリック変数の条件が指定できるなど探索戦略を柔軟に調整する手段が提供されている。

KLEEは LLVM プラットフォーム上で動作するため、LLVM bitcode にコンパイルできないプログラムには KLEE が適用できないという制約が存在する。SIL2LinuxMP では BusyBox に対して KLEE を使用してテストケース生成と同値分割テストを実施することが検討されている。KLEE はその性質上、プログラムの挙動が入力のみに依存して決定的であるものに対して有効で、外部からの割り込み、並列実行、タイミング等が挙動に影響する対象は適用が困難である。そのため Linux Kernel に対しては KLEE による解析は適さない。

3.1.3 反例検出

CPAChecker

CPAChecker [13] は C プログラムの各コードブロックへの到達可能性を検証するモデルチェックツールである。C ソースコードを入力として制御フローオートマトンを導出し、予め定義されたエラー状態へ遷移する可能性を解析する（図 3.3）。CPA は”Configurable Program Analysis”的頭文字であり、数あるプログラム状態遷移解析ツールの中でも、設定次第で一つのツールで複数用途の解析に使用可能であることが特徴である。例えば、メモリアクセス違反など特定の動的挙動に絞って解析を行う場合や、仕様検証のために正確性を重視したモデルチェックを行う場合などで、計算資源と正確性のトレードオフを考慮してメモリ割り当て量やタイムアウト時間などを制御することや、検証項目によってアルゴリズムを選択することができる。これらの検証アルゴリズム・パラメータセットは予め用意された設定が 180 存在しており、目的に従って既存の設定を選択使用することができる。CPAChecker は Linux Driver Verification Project (linuxtesting.org) で採用されており、Linux デバイスドライバに対して 150 件以上のバグ修正実績がある。

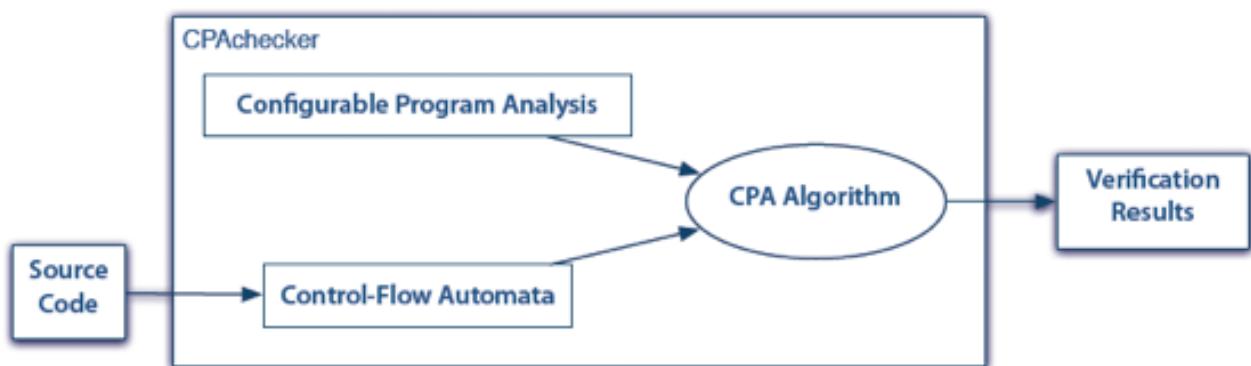


図 3.3 CPAChecker のワークフロー

以下に、プログラムのエラー状態への遷移可能性を CPAChecker によって解析する例を示す。

定義されたエラー状態に到達し得るミスがあるコード例

```
#include <stdio.h>

int main(void) {

    double x, y;

    // x で受け取った実数の絶対値を y に入れる
    scanf( "%lf" , &x);

    // 負数かどうかの判定を誤って-1 以下の条件としてしまった実装
    y = (x <= -1) ? (-1) * x : x;

    printf( "abs(%lf) = %lf\n" , x, y);

    // 計算した絶対値が非負でなかったらエラーとする
    if(y < 0) {
        goto ERROR;
    }

    return 0;

    // CPAChecker はソース中の”ERROR” ラベルか assert() を見つけて
    // そこへ遷移する条件（反例）を探す。このケースでは”ERROR” ラベルの代わりに
    // assert(y < 0); としても同じ検証ができる。
    ERROR:
    return 1;
}
```

CPAChecker はこのプログラムファイルと検証アルゴリズムの設定およびパラメータセットを読み込んで、検出されたエラー状態に遷移するための変数の状態を探索する。出力には、GraphViz フォーマット (.dot) による Control Flow Automaton と Abstract Reachability Tree が含まれる。CPAChecker の結果は Report Generator により web ブラウザで視覚的に表示できる形式で出力することが可能である（図 3.4）。

図 3.4 の Report Generator の出力右半分には関数毎の状態遷移図を表す Control Flow Automaton が表示されており、各エッジやノードをクリックすると該当する制御文や宣言などのコードにジャンプすることができる。ここで、検出されたエラーパスは赤く表示されており、このパスを通る際のコールスタックがレポートの左半分に表示されている。コールスタックの各行からも、該当する Control Flow Automaton のエッジまたはノードにジャンプすることができる。図 3.4 の左画面 (6) の次の行に表示さ

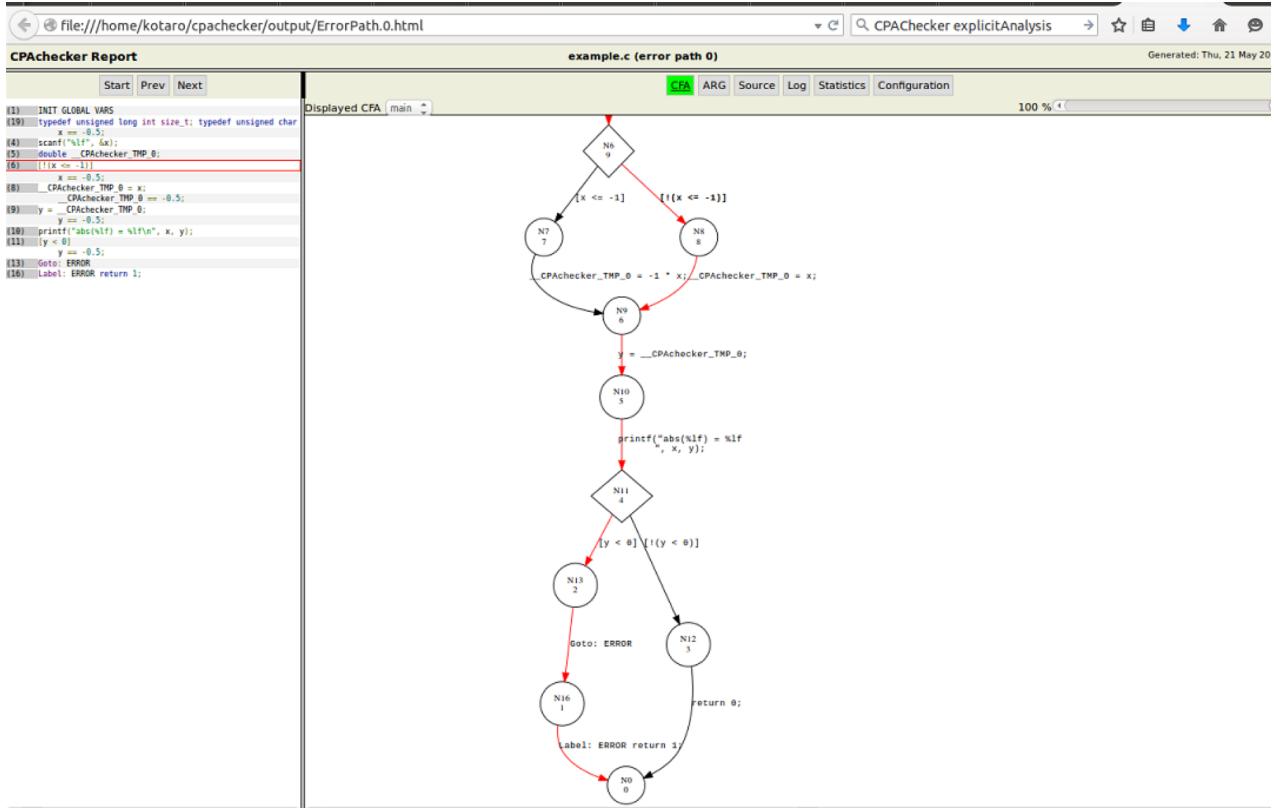


図 3.4 CPAchecker Report Generator の出力例

れている $x == -0.5$ が、探索された反例 (Counter Example) の具体値であり、これによってこのプログラムは $x == -0.5$ のときに ERROR: ブロックに遷移しプログラムが異常停止することが分かる。

以上の使用例に代表されるように、CPAchecker は予め定義されたエラー状態に到達し得るかどうかの判定 (Reachability Analysis) と、エラー状態に遷移し得る場合はその条件 (Counter Example) を具体的な変数状態で探索する。また検証アルゴリズムの設定によっては、NULL ポインタアクセスやバッファオーバーフローなどのメモリアクセス違反が発生し得るかどうかの判定も可能である。予めエラー状態がコード中に定義されていればそこに遷移する条件を自動的に検出できることから、GoogleTest 等のテスト駆動開発手法と上手に併用出来れば強力な回帰テストフレームワークを構築できる可能性がある。

CPAchecker が適用可能な C 言語表現は、ループ、配列、条件分岐、ポインタ、ヒープ、再帰、浮動小数点表示等と幅広く、CPAchecker は特に制御フローとメモリアクセスの安全性解析を得意とする。一方で、深くネストされたループやメモリを大量に消費する再帰構造を持つフローの解析は得意でなく、CPAchecker は無限ループの検出やデッドロックの検出を行うことができない。また、マルチスレッド等の並列実行構造を持つフローの解析にも対応していない。図 3.5 は、2015 年に行われた国際的なソフトウェア検証コンペティション Competition on Software Verification (SV-COMP) の総合結果 [11] である。SV-COMP では様々な種類の検証課題に対して各々検証プログラムのベンチマークが測定される。

CPAchecker は制御フローやメモリアクセスの安全性解析および Linux デバイスドライバの解析で圧倒的なパフォーマンスを示しており総合評価でもトップの成績を残している。ただし、先述したように深いループ・再帰の解析は不得意で停止性検証および並列実行の解析は対応していないため、これらの検証

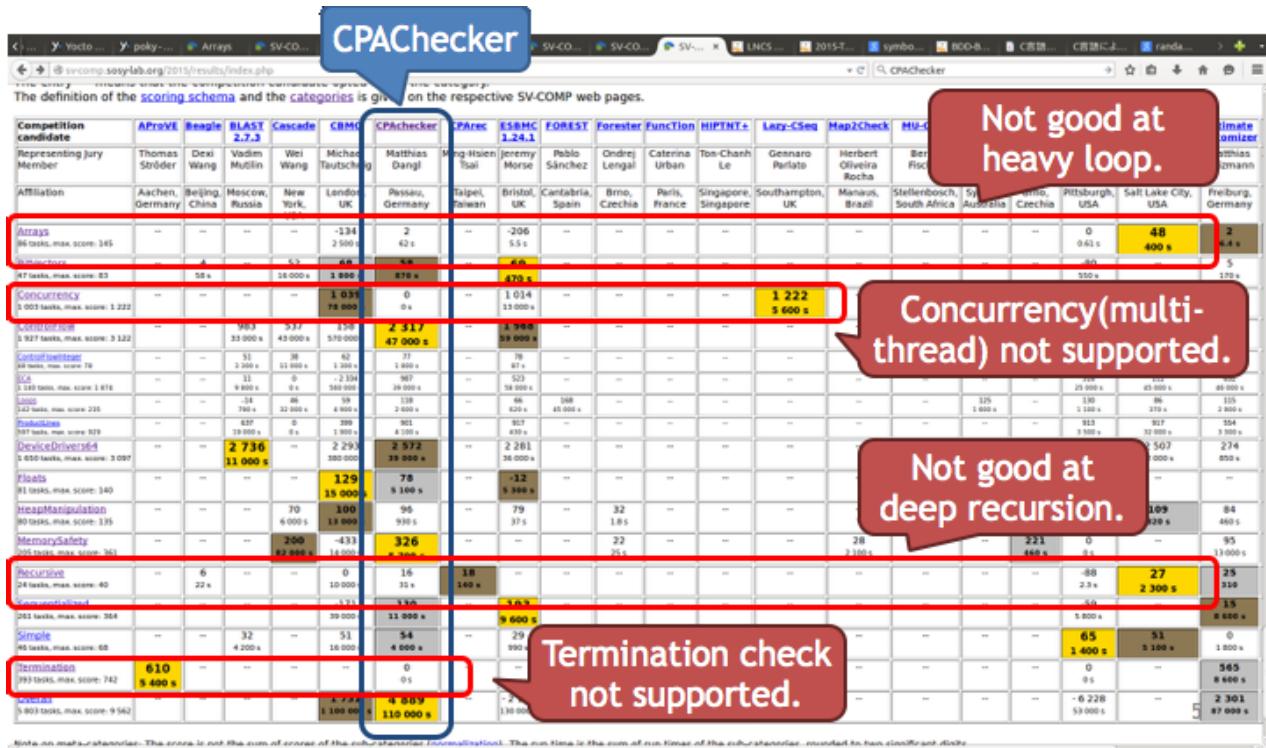


図 3.5 SV-COMP2015 の総合結果

課題に対しては他のツールが優位となっている。例えばプログラムの停止性検証 (termination check) では Automated Program Verification Environment (AProVE) [4] という停止性検証に特化した強力なツールが存在し、実際のデバイスドライバの検証にも大きな貢献をしてきている。停止性検証は古くから研究されている問題で、制御フローの解析問題とは独立にその分野自体に長い蓄積がある。全ての問題に対応する万能なツールを追求することは本質的ではなく、あくまでも目的とする検証対象に合致したツール・手法を選択しそれらの制約をよく把握した上で適切なツール・手法を使用することが重要である。

3.1.4 メトリクス測定

CodeViz

CodeViz [10] は C/C++ プログラムの関数呼び出し関係を視覚化する関数コールグラフ生成ツールである。SIL2LinuxMP では動的トレーシングツールの結果と組み合わせて呼び出し関係のカバレッジを測定する用途で使用することが検討されている（詳細は 3.2.2 項で記載）。関数コールグラフ生成ツールは、ソースコードを静的に解析してグラフを得るものと、実際にプログラムを動かしたときの挙動から動的にグラフを得るものとに大別できる。このうち CodeViz は静的に関数コールグラフを生成するツールである。静的コールグラフ生成ツールとしては他に GNU プロジェクトの cflow がある。cflow と比較して、CodeViz は関数ポインタを介した関数呼び出し関係も出力に反映できるという利点がある。図 3.6 は、Linux Kernel 4.3.3 ツリーの fs/read_write.c に対して vfs_read() 関数から先のコールグラフを生成した例である。



図 3.6 Linux Kernel fs/read_write.c の vfs_read() 関数コールグラフを CodeViz で生成した例

cflow と比較して、CodeViz の出力は次の点でより洗練されたものとなっている。

- `vfs_read()` と `__vfs_read()` を区別する (cflow は `__` 接頭辞が無視され両者が同一ノードになってしまふ)。
- 意図しないマクロ `likely()` や `unlikely()` のコールグラフへの出現が抑制されている (cflow では関数ノードとして出現してしまう)。
- `loff_t` など独自定義の型名が正しく処理されている (cflow では誤って `loff_t` が関数ノードとして出現してしまう)。

CodeViz は、コールグラフデータの生成手段として depn と ncc という二つの方法を提供している。depn は特定の GCC のバージョン (3.4.6 か 4.6.2) に対して専用のパッチを当てたものを使用してコールグラフデータを生成するため手順がやや複雑である。ncc [24] は特に GCC に対するパッチは必要なく手順が簡単である上、関数ポインタを介した呼び出し関係も表現することができる利点がある。SIL2LinuxMP では、手順の簡易さと関数ポインタを扱えるという利点から ncc を利用する検討が進んでいる。ncc はソースコードのブラウジング用途に開発されたコンパイラである。なお、グラフデータの視覚化は CodeViz 付属のツールを用いて画像を生成する方法と、ncc パッケージに付属する GraphViz データ (.dot) 変換ツールを用いる方法がある。関数コールグラフを利用した呼び出し関係のカバレッジメトリクス測定として SIL2LinuxMP で検討されている方法については 3.2.2 項で記載する。

syzkaller

syzkaller [30] は、Linux Kernel システムコールに対する fuzzer プログラムである。Fuzzer とは、大量のランダムデータ・ランダムテストケースを生成・実行することで検査対象プログラムのミスや脆弱性を見つけるテストプログラムの総称を言う。[3.1.2](#) 項で触れた Csmith はコンパイラに対する fuzzer である。syzkaller はシステムコールに対するカバレッジを指標にテストケースを自動生成・実行する。これを利用すると、ターゲットシステムの debugfs からシステムコールに対するテストカバレッジ情報を取得できたり、クラッシュ情報やテスト結果を取り出して開発ホスト側の web インタフェースでサマリを表示したりということが可能になる（図 [3.7](#)）。

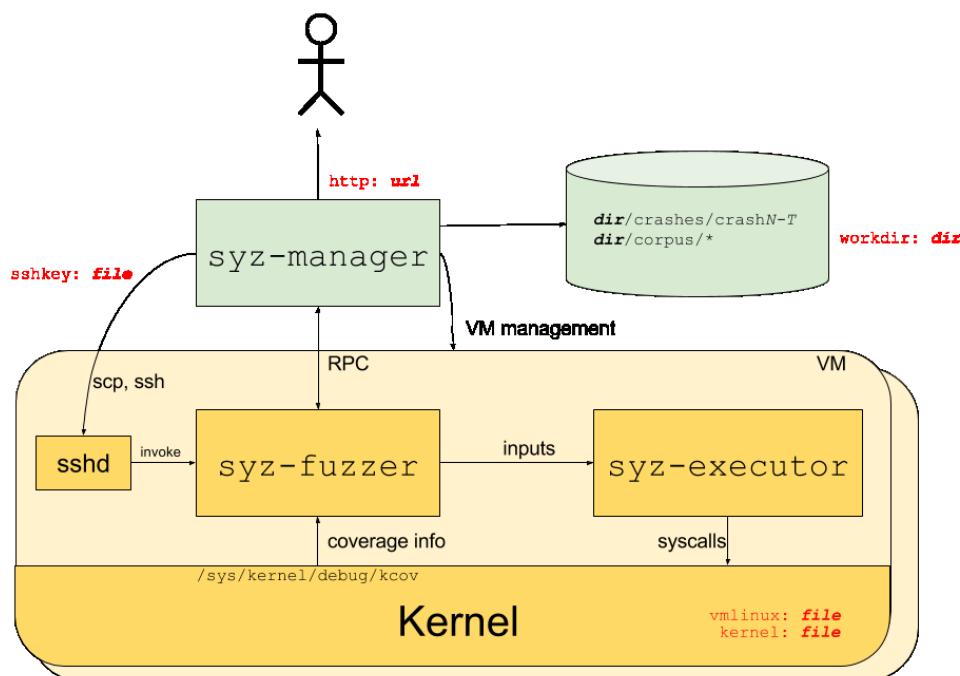


図 3.7 syzkaller システムの全体アーキテクチャ

syzkaller は 2015 年 10 月に開発が始まった新しいプロジェクトであるため成熟度は十分でなく、現在サポートされるプラットフォームは QEMU のみで他の仮想マシンや物理ターゲットボードは未サポートである。また現状は導入の前提条件として最新の GCC を自前でビルドする必要があったり、Linux Kernel に特別にパッチを当てる必要があったりと手順に難がある。SIL2LinuxMP では Linux Kernel システムコールのテストカバレッジ向上とメトリクス測定のために syzkaller 利用すること想定して調査が継続されている。

3.1.5 バグ選別

3.1.1 項で述べた通り、Coccinelle はそれ単独で用いると大量の False-Positive を生成してしまうため、真のバグをいかに選別するかが効率的なバグ検出にとって重要となる。

Herodotos

Herodotos [21] は、Coccinelle の False-Positive を抑制する目的で Coccinelle と同じ INRIA が開発しているツールで、特定のバグパターンの出現傾向を世代間にわたって解析することができる。Semantic Patch で一般化したバグパターン各々についてバグのトレンドと残存期間などを視覚化することで、そのプロジェクトの開発プロセスの安定性や健全性を示す根拠の一つとして利用することが SIL2LinuxMP では検討されている。図 3.8 は、”badzero”というバグパターンが Linux、Wine、VLC、OpenSSL の各プロジェクトについて世代ごとにどの程度残存していたかをグラフで示したものである。

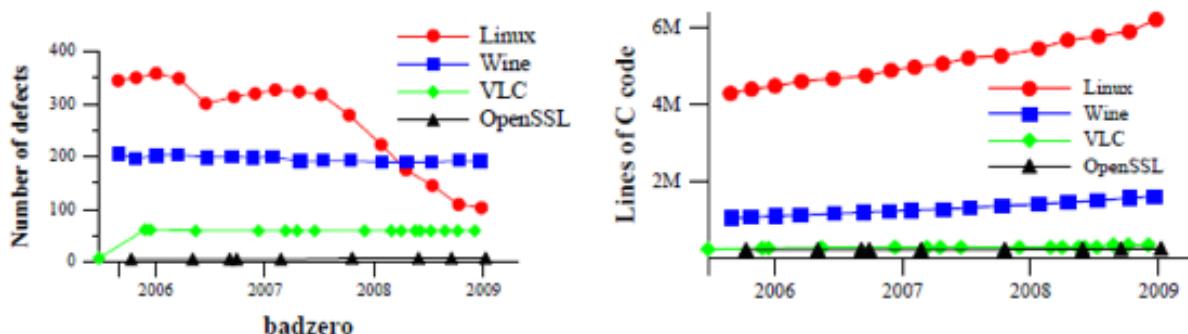


図 3.8 各プロジェクトの各世代に対する”badzero”バグの存在数（左図）と各プロジェクトの各世代でのコードサイズ（右図） <https://hal.inria.fr/inria-00406306/PDF/RR-6984.pdf> から引用

図 3.8 から、Linux Kernel はその成長に伴って badzero バグが積極的に改修されているのに対して、その他のプロジェクトでは同種のバグがほぼ放置されたままとなっていることが分かる。

図 3.9 に Herodotos ワークフローの概要を示す。Herodotos は、解析対象となる特定パターンの出現場所が記録された②”pattern occurrence reports”と、解析対象となるソフトウェアバージョン間の差分情報③”code changes”を入力とする。ここで、②”pattern occurrence reports”は Coccinelle の出力で、③”code changes”は GNU diff の出力である。図 3.9 中の”code pattern”は Semantic Patch、“pattern matching tool”は Coccinelle、“diff tool”は GNU diff を意味する。”tracking environment”に相当する箇所は、解析対象となるバグパターンを記載した Semantic Patch と解析対象の各バージョンのソフトウェアを含めて自前で用意する必要がある。④で②”pattern occurrence reports”と③”code changes”を受け取った Herodotos は、入力された特定パターンを入力されたバージョン間にまたがって自動で関係付ける。ここで、入力されたパターンを Herodotos が自動的に関係付けられなかったコードについては人間が手動で”SAME”か”UNRELATED”かを判断してラベル付けを行う。新たに入力されたパター

ンについては人間が手動で”BUG”か”False-Positive”かを判断してラベル付けを行う。つまり最初は全て人間が特定パターンが”BUG”か”False-Positive”かを Herodotos に教えるところからスタートする。このプロセスを回すことによって、過去のバージョンで False-Positive とラベル付けされ、かつ新しいバージョンでも同じ箇所に同じパターンが残っていると判断されたものについては自動的に False-Positive のラベルが付与されることになる。最終的に Herodotos は特定パターンの時間軸（バージョン毎）における統計情報とグラフを出力する。

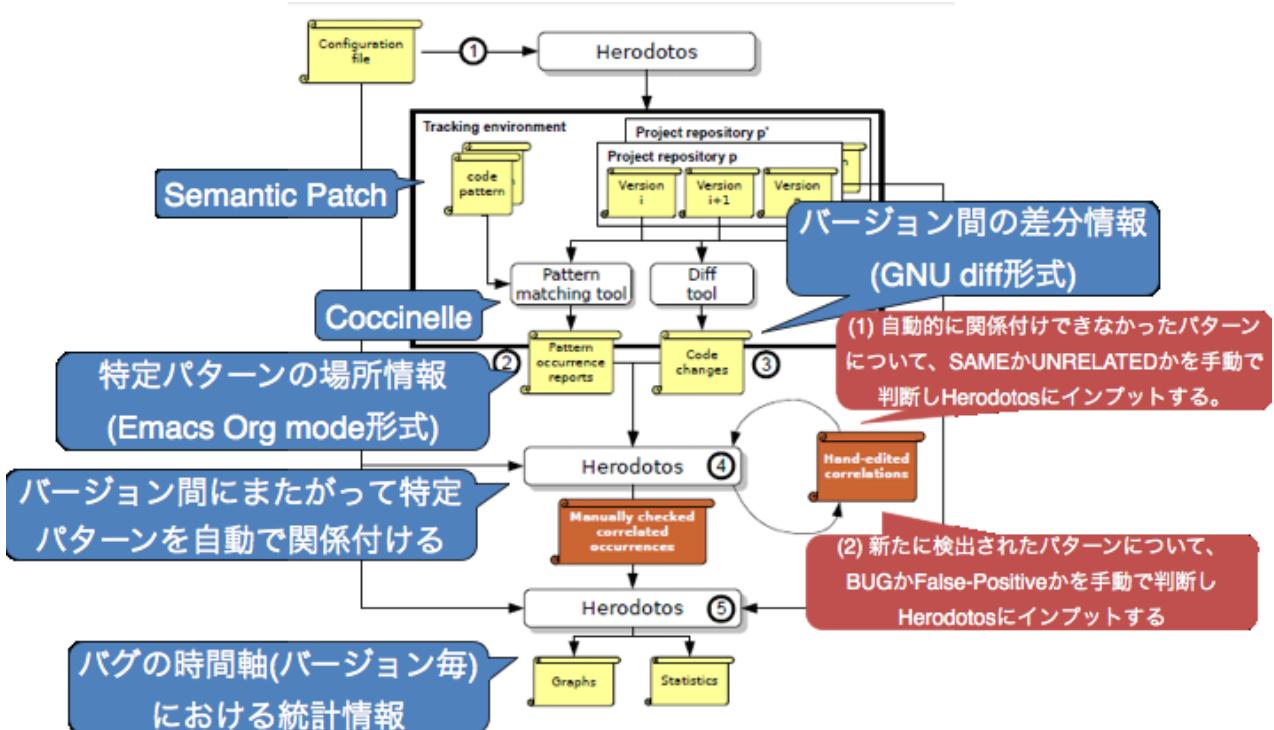


図 3.9 Herodotos のワークフロー

なお、Coccinelle に限らず一般に False-Positive を抑制する方法としては、使用する kernel config に基づいて無効な #ifdef コードブロックを削除する前処理を行うことでビルド対象だけからなるソースコードを抽出し、検査対象となるソースコードそのものを絞るという方法も有りうる。この手法は 3.4 節で詳述する。

Prequel

Prequel はセマンティックパターンサーチをコミット履歴（パッチ）に対して行うツールで、Coccinelle と Herodotos と同じく INRIA が開発している。Coccinelle がソースコード空間に対してセマンティックパターンサーチを行うのに対して、Prequel はサーチ対象を時間軸であるコミット履歴（パッチ）に拡張する。Prequel は開発中のためまだツールの実態は存在しないが、主に次の目的で利用されることが想定されている。

- ある修正を行いたい場合に、過去に似たような修正が行われていないかを探すことでの適切な修正方法や副作用を調査する。
- 過去に行われた修正で現在のコードの別の箇所にも同じ修正を適用するべきパターンが残存していないかを探し、同じ問題が繰返し発生することを防止する。
- コミットログを解析し、各開発者に対するメトリクス（スキルレベルや危険なコードを書く可能性など）を測る。

bugspots

SIL2LinuxMP で取り上げられた手法ではないが、Prequel と関連してコミット履歴を元にバグ予測を行う手法に次のようなものが知られている。Google が 2011 年に公開したバグ予測アルゴリズム [18] によると、「バグ修正が最近頻繁にコミットされている箇所ほど残存バグがある可能性が高い」という経験則がある。Google では毎日大量のコードコミットがあり人手によるレビューを全ての変更に対して十分に行うことは現実的に不可能であるため、バグがある可能性が高い”hot spot”に集中してレビューを行う必要がある。そのような hot spot を特定するために、trial & error の末に導かれた式が (3.1) である。

$$Score = \sum_{i=0}^n \frac{1}{1 + e^{-12t_i + 12}} \quad (3.1)$$

ここで、 n はバグ修正に関連するコミットの数、 t_i は i 番目のバグ修正コミットのタイムスタンプでこの値は 0 ~ 1 で正規化されている。0 はソースコードベースで最初のコミットがあった時点を表し、1 は現時点を表す。つまり式 (3.1) を評価する時点によって t_i の値は変動する。式 (3.1) は新しいバグ修正ほど大きく重み付けされ、過去のバグ修正ほど小さく評価されるようスケーリングされている。これをグラフに描くと図 3.10 のようになる。

式 (3.1) の $Score$ が一定の閾値を超える箇所に絞って集中的にコードレビューやコストのかかる検証を行うことで、ソフトウェアの品質を効率良く向上させることができる。これはバグ修正のコミットタイミング情報のみを利用するごく単純な方法であるが、他のバグ予測アルゴリズムを抑えてバグの出現傾向を良く近似しているという。bugspot [5] はこの hot spot 特定技法の Ruby によるオープンソース実装である。

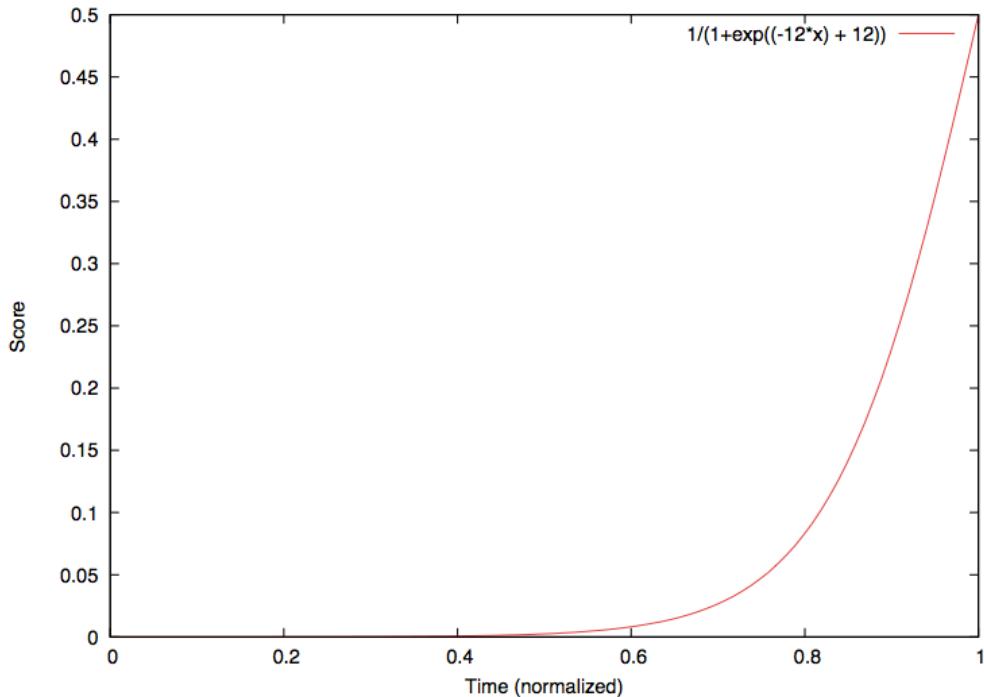


図 3.10 バグが残存する可能性が高い hot spot を評価する式 (3.1) のグラフ

3.1.6 プロジェクト管理

機能安全対応の開発においては開発対象のソフトウェアそのものに対する正当性検証に加えて、要求と設計のトレーサビリティなどマネジメント観点でのエビデンスを体系的に残し、開発プロセスが健全であることを認証機関に説明することも求められる。次に SIL2LinuxMP で採用が検討されているプロジェクト管理ツールを述べる。

OGSN

Goal Structuring Notation (GSN) は、達成すべき目的・性質・品質についてそのゴールを実現するための戦略・道筋・方法・思考を可視化するためのアシュアランスケース記法の一つである。GSN は Evidence Chain を記述するための国際的な標準表記法であり ISO26262 で採用されている。機能安全対応の際は特に Safety Case Concept について、思考の過程を共有し系統立てて議論を進め「なぜその方法で目的が達成できるのか」を認証機関に説明するために GSN 表記が用いられる。

[参考ブログエントリ：GSN\(Goal Structuring Notation\) 解説 \[20\]](#)

GSN 表記は、長方形で示されたゴールが複数のサブゴールにブレークダウンされ、それらが最終的に円で示されたエビデンスまたはソリューションで支えられる構造を取る。ゴール（長方形）とエビデンス（円）の間には戦略が平行四辺形で記述され、その戦略の根拠や仮定が楕円で、ゴールを目標として成立させている前提条件が角丸長方形で表現される（図 3.11）。

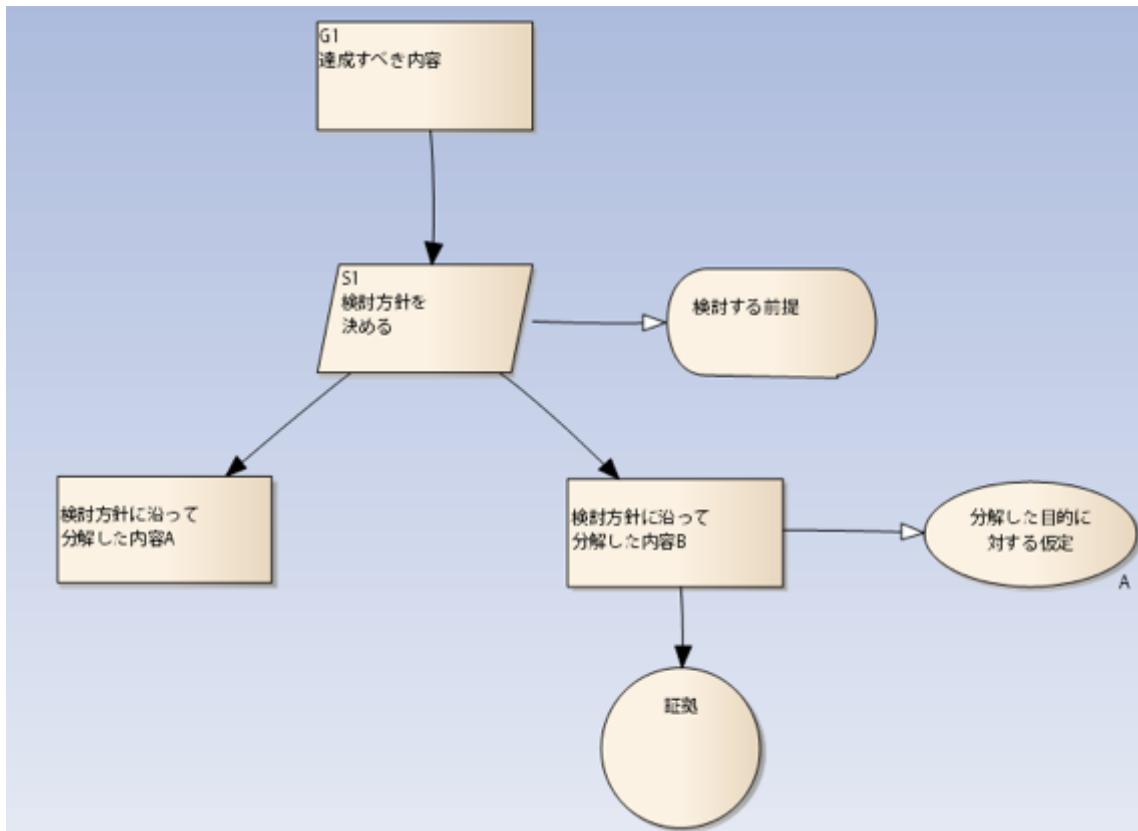


図 3.11 GSN 表記の基本的な構造

SIL2LinuxMP では Safety Case を GSN 表記で記述するためのエディタとして既存のツールが利用可能であるかが調査されたが、最終的に SIL2LinuxMP プロジェクトで独自に OSADL GSN Editor (OGSN) を開発することとなった。IEC61508-3 7.4.4.3 には、off-line 開発ツールの選定条件および選定根拠が明示されることとの記述がある。これに従い SIL2LinuxMP では GSN エディタに求めるマネジメント要件と機能要件として次の項目を挙げた。

- マネジメント要件
 - プロジェクトの初めから最後まで利用できることが保証されていること
 - 様々な開発プラットフォーム上で利用可能であること
 - 将来にわたっても利用できる実行できること
 - オープンソースライセンスであること
- 機能要件
 - GSN コミュニティが定める [GSN 標準記法 \[19\]](#) Part1 で定義された GSN 基本要素に対応していること
 - [GSN 標準記法 \[19\]](#) Annexes A1 と B1 の特に”Away Goal”表記に対応していること

上記で定めた要件をクライテリアとして、3つの GSN エディタ DCASE、ACEdit、AdvoCATE が調査された。GSN エディタには商用で利用できるものが多数存在するが、それらはオープンソースライセンスでないため選定候補にはならない。調査を行った既存の GSN ツールと各々の特徴概要は次の通りであった。

- DCASE [15]
 - Eclipse 上で動く Java 実装の拡張 GSN エディタ
 - 厳密な GSN standard に従っていない独自の表現がある
 - 保守が止まつていて 2010 年の Eclipse でないと使えない
- Assurance Case Editor (ACEdit) [3]
 - Eclipse 上で動く Java 実装の拡張 GSN エディタ
 - DCASE よりも GSN 標準に従っている。
 - Eclipse プラグインのインストールに難あり
- AdvoCATE [2]
 - Eclipse 上で動く Java 実装の拡張 GSN エディタ
 - グラフ要素の形が GSN 標準からかけ離れている
 - Eclipse プラグインのインストールに難あり
 - ホームページの最終更新が 2011 年

調査の結果、各ツールともメンテナンス状況が十分でないことと Eclipse に付随するバージョン・プラグイン依存関係が複雑であることから、SIL2LinuxMP では既存ツールから適切なものを選定する方針を改め、独自にシンプルで必要最小限の GSN エディタ OGSN を開発することとなった。OGSN は 120 行程度の Python スクリプトであり、必要最小限の依存ライブラリ (ConfigParse, PyGraphviz) を持ち、GSN グラフ表現では GraphViz を使用して自動的にノード・エッジを配置する仕組みとなっている。OGSN は SIL2LinuxMP コアメンバーの OpenTech がプロトタイプを開発中で、今後は次の課題を中心に実装が進められる予定である。

- GSN 標準記法の違反・例外チェック (ループ検出等) 作りこみ
- GSN 標準で定められた記法の順次適用
- GSN グラフ表示形式の洗練、Sub Tree の導入
- 複数分割ファイル入力のサポート

rmToo

SIL2LinuxMP プロジェクトにおいて、開発履歴のトレーサビリティは Git で、目標 (ゴール) からエビデンス・ソリューションへのトレーサビリティは OGSN で確保されるが、これのみでは安全要求の開発履歴や要求間および要求と GSN で表現した目標やソリューションとのトレーサビリティを取ることができない。そこで SIL2LinuxMP では GSN エディタと同様にシンプルなオープンソースの要求管理ツールとして Requirement Management Tool (rmToo) [26] の使用が検討されている。

rmToo は Python で実装されたシンプルなコマンドラインツールであり、Plain Text で書かれた要求事項から HTML または PDF 形式でのドキュメントや要求項目間の依存関係グラフを出力する。rmToo には GUI や専用のエディタは付随しない。テキストベースの CUI ツールであるという特性は LATEX と親和性が高く、SIL2LinuxMP の System Requirement Specification (SRS) 文書は LATEX に OGSN で描いた図と rmToo が併用されて要求間のトレーサビリティが考慮された構成となっている。ツールの成

熟度は高いが、rmToo がただ一人の開発者によって書かれていることと、GitHub 上の更新が 2012 年 12 月で止まっておりメンテナンス状況が不透明であることが懸念点として挙げられている。

3.1.7 その他

本節で詳しく述べることはしないものの、SIL2LinuxMP で利用することが検討されているツール、または OSS・Linux 開発において標準的に用いられているツールとして次のようなものがある。

トレーシング・プロファイリング： ftrace, kprobes, perf

カバレッジメトリクス測定： gcov/lcov

ベンチマーкиング： lmbench, hackbench, crashme

新たな開発ツールの採用を検討するときは、3.1.6 項の GSN エディタ選定で行ったようにツールに求める要件とクライテリアを定めて、各ツール候補についてアセスメントを行った上で適切なツールを選ぶ。または要件を満たす既存ツールがなければ独自に新規開発を行う判断をする。独自に新規開発を行う場合は、IEC61508 が示す 3 通りの規格準拠方法(4 ページ)のうち 1_S (初めから最後まで全て厳格に品質管理されたプロセスの下で開発・評価を行う) に従うことになる。

3.2 SIL2LinuxMP の戦略と規格準拠プロセスの理解

3.1節では、SIL2LinuxMP プロジェクトでの利用が検討されているツール類のうち代表的なものを述べた。本節では、機能安全対応開発においてツールセットを選択するための戦略と、既存ツールを組み合わせて新たなエビデンスを得る方法について SIL2LinuxMP で検討されているアイディアを記載する。

3.2.1 SIL2LinuxMP のツール選定戦略

SIL2LinuxMP の開発基本方針は Route 3_S: assessment of non-compliant development（規格非適合箇所に対して対策を行い適合と証明するに十分な根拠を与える）である。このことは、従来の開発プロセスにおいて「設計・実装」に相当するフェーズが既存ソフトウェアの「選定」に置き換わるを意味している（図 3.12）。

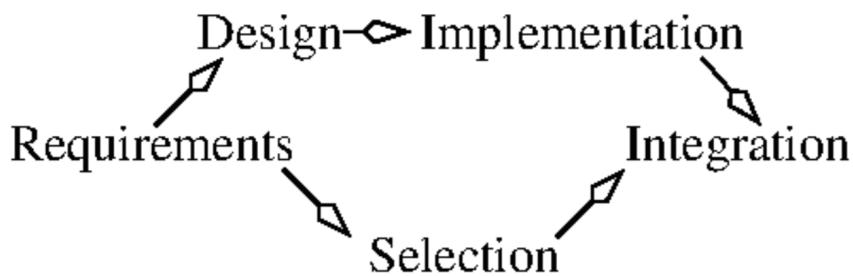


図 3.12 既存ソフトウェアの選定プロセスが中心となる開発プロセスへの変化

選定フェーズの入り口ではソフトウェアに対する要求として選定クライテリアを定め、出口では定めたクライテリアに対する既存ソフトウェアのアセスメント結果を用意する。既存ソフトウェアそれ自体が十分な品質・安全性を保証できない場合はそれを補うエビデンスを生成できる手段やツールを新たに導入する。新たなツールの導入に関して特に考慮すべき事項は以下の通りである。

- 非機能要件（実行環境・言語・依存ツール・使用方法）
- 機能要件（最低限備えるべき機能・あれば良い機能）
- カスタマイズの自由度
- コミュニティの成熟度と実績

導入するツール自体についてもそれを採用する妥当性を認証機関に証明することが求められる。ここでツール自体が多機能かつ大規模で複雑であったり、実行環境が特殊であったり、依存物が多くたりすると精査しなければいけない対象と調査コストが現実的でない規模にまで膨らんでしまう。そのため機能安全開発で使用するツールとしては、妥当性を証明できるほど十分に小さい構成と簡単な依存関係を持ち、かつ一般に広く認知されていて既に実績がある技術を使用したものが有利である。

また新しいツール導入の際は、既に使用している開発ツールセットとの整合性もツール選定の基準となる。同じ開発プラットフォーム上で動作するか、依存物の競合を起こさないか、バージョンの組み合わせは問題ないかなどを精査し、全体の開発ツールセットが「一貫した整合性を持つ必要最小限の構成」となるよう考慮する。必要最小構成のツールセットを実現するためには、例えば使用しない機能を無効にすることでその部分の妥当性検証を省略する戦略が考えられる。その場合は設定をオープンに操作できるカスタマイズ自由度があるツールほど有利である。

スケールする検証フレームワークを実現するための戦略

SIL2LinuxMP に特有の事情として、認証対象がオープンソースソフトウェアであるという点がある。このことは、認証対象の妥当性を示すためのエビデンスを生成する検証ツールもオープンソースでなければならぬことを意味する。vendor lock-in を回避するため、かつ認証対象ソフトウェアのバージョンアップに追従するため、それらに対する検証は全てオープンな技術に基づいて開発者自身が実施できるものである必要がある。このような要件に対応するためには何らかの検証自動化の工夫が不可欠である。加えて、過去に実施したテストケースなどの資産を漸増的に繰返し再適用でき、さらにソースコードのアップデートに基づいて自動的にテストケースが追加生成されることでその検証フレームワークを回せば回すほど自動的に品質を裏付けるエビデンスが蓄積されていくような仕組みを構築できれば、開発プロセスをスケールすることができる。[3.1](#) 節で述べたツールは各々、このような「スケールする検証フレームワーク」の一要素となる特性を持っている。Coccinelle は過去に明らかになったバグを Semantic Patch で一般化・蓄積し回帰テストに用いることで同種のバグが新たに発生することを未然に防ぐことができる。Csmith はランダムテストケースを生成し続けることでテスト母数を増やし品質の裏付けとなるエビデンスを蓄積することができ、蓄積されたテストケースはソフトウェアのバージョンアップに対する回帰テストとして繰返し適用できる。そして SIL2LinuxMP では新たに fuzzer の一種である syzkaller の調査検討が進められているように、「スケールする検証フレームワーク」を構築するためにはテストケースを自動生成できてそれを蓄積し回帰テストで繰返し適用できるような性質を持つ技術要素が特に有効であると考えられる。

しかし図 [3.1](#) で述べた通り、品質管理ライフサイクルにおいてはテスト実行やテストケース生成にとどまらず、バグ抽出 (False-Positive フィルタリング)、バグレポート生成、バグ管理、メトリクス測定・テストレポート生成など様々なタスクを実施する必要がある。「スケールする検証フレームワーク」を構築するためには、これらのタスクを含めた品質保証エコシステム全体を自動化する工夫が必要となる。特に問題となると考えられるのは「テストケース生成」と対になる「バグ抽出」の自動化である。Coccinelle と Csmith の項目で述べた通り、これらは出力に大量の False-Positive が含まれるという共通の課題を持つ。INRIA が Coccinelle に対する False-Positive 抑制策として Herodotos や Prequel を提案しているが、これらはまだ人手の介入度合いが多く実績も少ない。検証ツール全体を見渡しても、fuzzer に代表されるテストケース生成自動化のためのツール・手法は豊富にあるものの、大量のテスト結果から真のバグをシステムティックに抽出する技法はほぼ存在しないように思われる。テストケース生成を自動化した結果大量の False-Positive が出力され、その選別のために人手が必要である限り「スケールする検証フレームワーク」は実現できない。テストケースを継続的に管理する仕組み構築に対して、テスト結果から真のバグを選別する何らかのバグトリアージの仕組みが不可欠である（図 [3.13](#)）。



図 3.13 品質保証エコシステムにおいて対となるテストケース生成部分とバグ抽出部分

バグトリアージという言葉は一般には「大量の不具合から優先して対処すべき案件を選別する」ことを意味するが、図 3.13 では「大量の False-Positive を含む不具合から真に不具合である案件を選別する」という拡張した意味で用いている。テスト対象に応じて既存ツールから適切なテストケース生成ツールを選定するように、バグ抽出も対応の仕方はテスト結果の形式次第となる。ただしここで場当たり的な False-Positive 選別ではなく、bugspot のように確率統計情報を用いて真のバグが残存する可能性が高い箇所を絞り込むことや、そもそも False-Positive を出力させない工夫をテストケース生成側で施すことなど、システムティックなバグトリアージ手法を確立する工夫が必要である。具体的なバグトリアージ手法は今後の調査研究課題である。

3.2.2 統合検証フレームワーク：DB4SIL2

IEC61508-3 Table B.2 - Dynamic analysis and testing (図 3.14) の 7a~7d では、各種カバレッジ 100% を達成することが Safety Integrity Level 2 (SIL2) で Recommended (R) または Highly Recommended (HR) とされている。ここで、カバレッジ測定のための技法は対象となるソフトウェアの特性に従って適切に選択されなければならない。Table B.2 の通り 100% カバレッジが達成できない場合はカバーできない箇所に対する合理的な説明を与えるか、カバーできない部分を特定し何らかの代替手段で補うことが求められている。

一般にカバレッジを測定するテスト手法には、対象となる OSS ソフトウェアを開発するコミュニティで標準で使われている手法・ツールを用いる。例えば Linux Kernel であれば LTP や POSIX test suite を利用することをまず検討する。その上で、既存の手法・ツールではカバーできない領域を特定し、そのようなコード領域へ何らかの対策を行う。カバーされないコード領域への対策としては例えば次の 1, 2

(Referenced by Tables A.5 and A.9)

Technique/Measure *		Ref	SIL 1	SIL 2	SIL 3	SIL 4
1	Test case execution from boundary value analysis	C.5.4	R	HR	HR	HR
2	Test case execution from error guessing	C.5.5	R	R	R	R
3	Test case execution from error seeding	C.5.6	---	R	R	R
4	Test case execution from model-based test case generation	C.5.27	R	R	HR	HR
5	Performance modelling	C.5.20	R	R	R	HR
6	Equivalence classes and input partition testing	C.5.7	R	R	R	HR
7a	Structural test coverage (entry points) 100 % **	C.5.8	HR	HR	HR	HR
7b	Structural test coverage (statements) 100 %**	C.5.8	R	HR	HR	HR
7c	Structural test coverage (branches) 100 %**	C.5.8	R	R	HR	HR
7d	Structural test coverage (conditions, MC/DC) 100 %**	C.5.8	R	R	R	HR
NOTE 1 The analysis for the test cases is at the subsystem level and is based on the specification and/or the specification and the code.						
NOTE 2 See Table C.12.						
NOTE 3 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.						
* Appropriate techniques/measures shall be selected according to the safety integrity level.						
** Where 100 % coverage cannot be achieved (e.g. statement coverage of defensive code), an appropriate explanation should be given.						

図 3.14 IEC61508-3 Table B.2 - Dynamic analysis and testing

のような戦略が考えられる。

1. 該当コードが実行され得ないことを示してテスト不要であることを証明する
2. 該当コードをテストするテストケースを何らかの手段で生成する

戦略 1（テスト不要箇所の特定）の具体例としては、コンパイル対象とならないデッドコードを特定してそもそも検証対象から外す方法があり得る。このデッドコードを除外する手法については 3.4 節で詳述する。または、CPAChecker による形式検証手法により到達し得ないコードを特定して検証対象から外す根拠とすることも考えられる。戦略 2（テストケースの補完）については、例えば KLEE による Symbolic Execution ツールで必要なカバレッジを達成するテストケースを（半）自動生成することが考えられる。このように、コード内部構造の理解に基づいて Black Box テストのパフォーマンスを向上させる戦略は GrayBox テストと呼ばれることがある。SIL2LinuxMP プロジェクトでは、GrayBox テストをはじめ OSS・Linux の機能安全対応に必要な検証技法を実装するための統合検証フレームワークとして Database for SIL2 (DB4SIL2) を開発している。DB4SIL2 の開発では、予め定められたクライテリアによって選定された検証ツールセットが統合され、ツール同士が入出力データをやり取りし互いにエビデンスを補完しあう形で図 3.13 のような品質保証エコシステムを構築することを目指している。

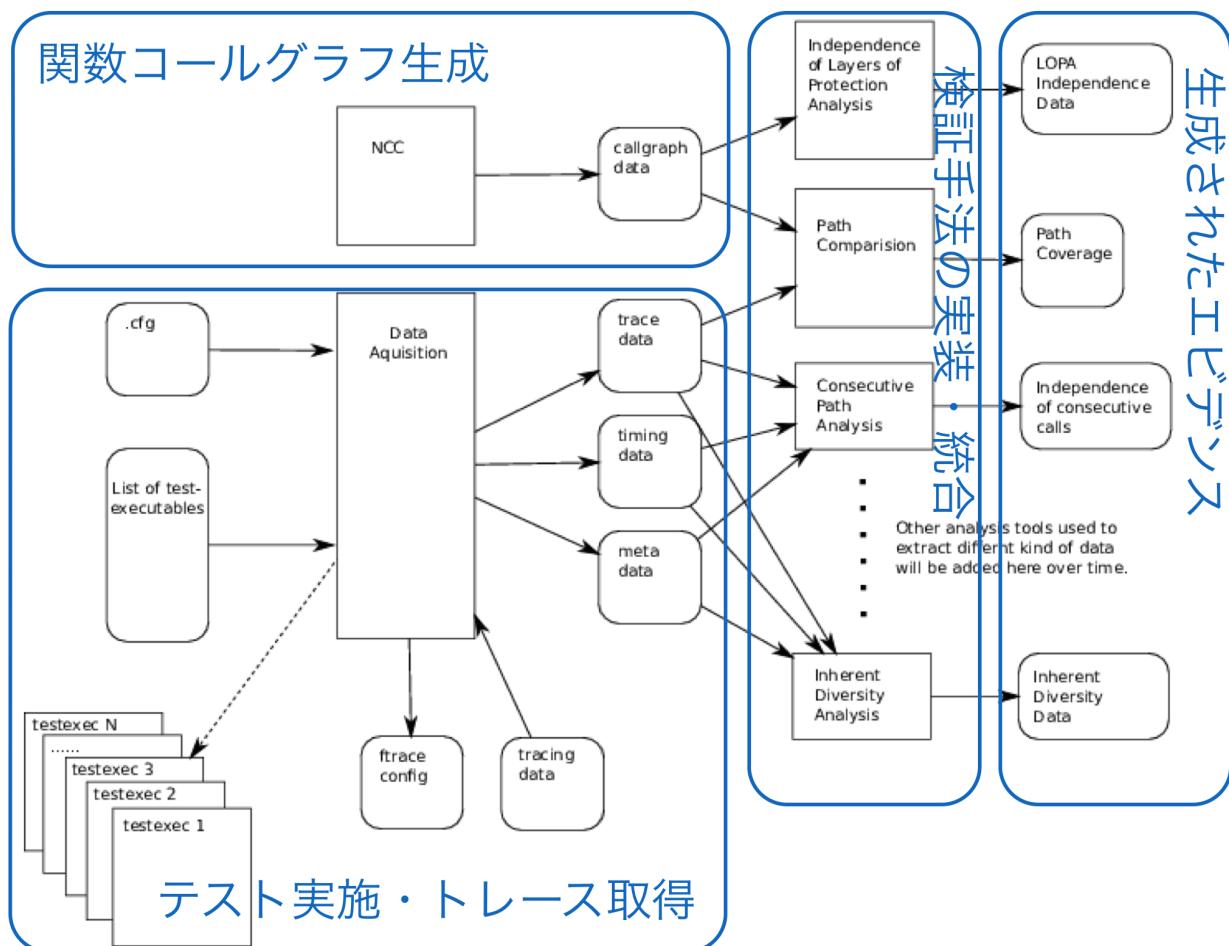


図 3.15 DB4SIL2 の構想：機能安全認証に必要なエビデンスを生成する仕組みを実装・統合する

図 3.15 は現時点での DB4SIL2 の構想である。DB4SIL2 は関数コールグラフやトレーサなど既存ツールで得られるデータを入力とし、機能安全認証プロセスで求められる様々なエビデンスを生成する。以降では必要なエビデンスを得るために DB4SIL2 開発で具体的に検討されている検証手法を述べる。

関数コールグラフとトレーサを用いたカバレッジ解析手法

GrayBox テストを行うためには、実施されたテストでカバーされていないコード領域を特定することが前提として必要である。本章では、関数コールグラフとトレーサを用いてカバレッジ解析を行う手法として DB4SIL2 開発で検討されている戦略を述べる。

図 3.16 は、事前にソースコードの静的解析によって得られた関数コールグラフと、テストによって得られた動的トレース結果を付き合わせることでパスカバレッジに関するメトリクス解析を行うワークフローである。ここで、静的関数コールグラフを得るために 3.1.4 項で記載した CodeViz を使用する。CodeViz の ncc を使い関数ポインタを介した呼び出し関係までを静的に解析することで、対象とするソースコードにおいて実行される可能性のある関数呼び出しパスを全てリストすることができる。動的トレースの取得には Function Tracer (ftrace) を用いる。ftrace は、kernel 空間の動的挙動に対するデバッグや

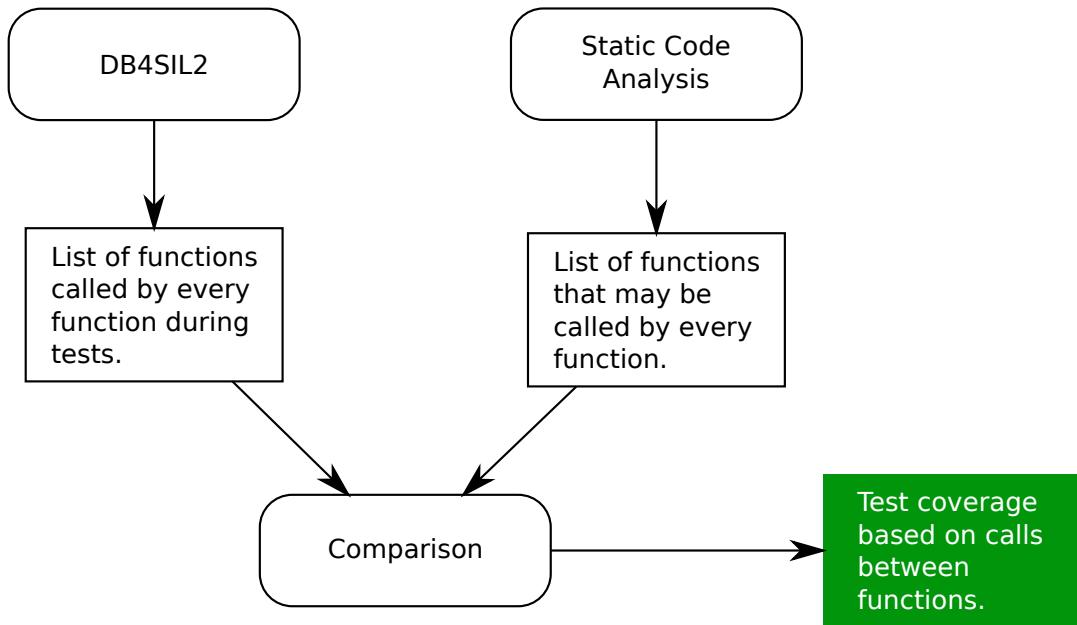


図 3.16 関数コールグラフとトレーサを用いたパスカバレッジ測定方法

障害・性能解析ツールとして広く使われている Linux Kernel の機能であり、予め埋め込まれたトレースポイントを契機として関数呼び出し履歴と各々の関数の実行時間を低負荷で記録することができる。ここで、実行され得る関数呼び出し関係を CodeViz を使いソースコードを基に全て洗い出した上で、そのソースコードでビルドしたソフトウェアで任意の動的テストを行ったときの関数呼び出し履歴を ftrace で取得することを考える。これによって得られた静的関数コールグラフと動的関数コールグラフを比較した結果、静的グラフに存在するが動的グラフには出現しない関数呼び出しパスが特定されれば、そのパスが実施されたテストでカバーされていない部分であると言うことができる。また、その比較解析結果から関数呼び出しパスについてのカバレッジメトリクスを測定することができる。

例えば、システムコール `SyS_sched_getscheduler()` を契機に呼ばれる可能性のある関数は CodeViz の関数コールグラフ導出により明らかになる（図 3.17）。ここで、あるテストを行った結果 ftrace によるトレースが図 3.17 中の Trace1, Trace2 のように得られたとする。Trace1, Trace2 中で現れた関数呼び出しパスをコールグラフ中で探すと、コールグラフで ✓ を付けたエッジが見つかる。✓ を付与していないエッジがカバーされていない関数呼び出しパスに相当する。

本項で述べた解析手法によって得られ得た静的・動的関数コールグラフは、ソースコードに変更が入った際にその全体への影響を調べるインパクトアナリシスにとっても有用なデータとなる。変更が加えられた関数を直接・間接的に呼んでいる関数、または変更箇所から呼ばれている関数が既に明らかであれば、変更の全体への影響を調べるために関数コールグラフ上変更箇所と関係を持つコード領域のみに注目してレビューや回帰テストを実施すればよい。データを介したやり取りや、イベント・割り込みが関わる直接的な関数呼び出しでない非同期挙動の分析はこの限りではないが、インパクトアナリシスの範囲を考えるための一つの指標として用いることができる。

以上のような関数コールグラフとトレーサを用いたカバレッジ測定手法およびインパクト解析手法は、ftrace の性質上適用できる範囲がカーネル空間のみに制約される。しかし SIL2LinuxMP プラット

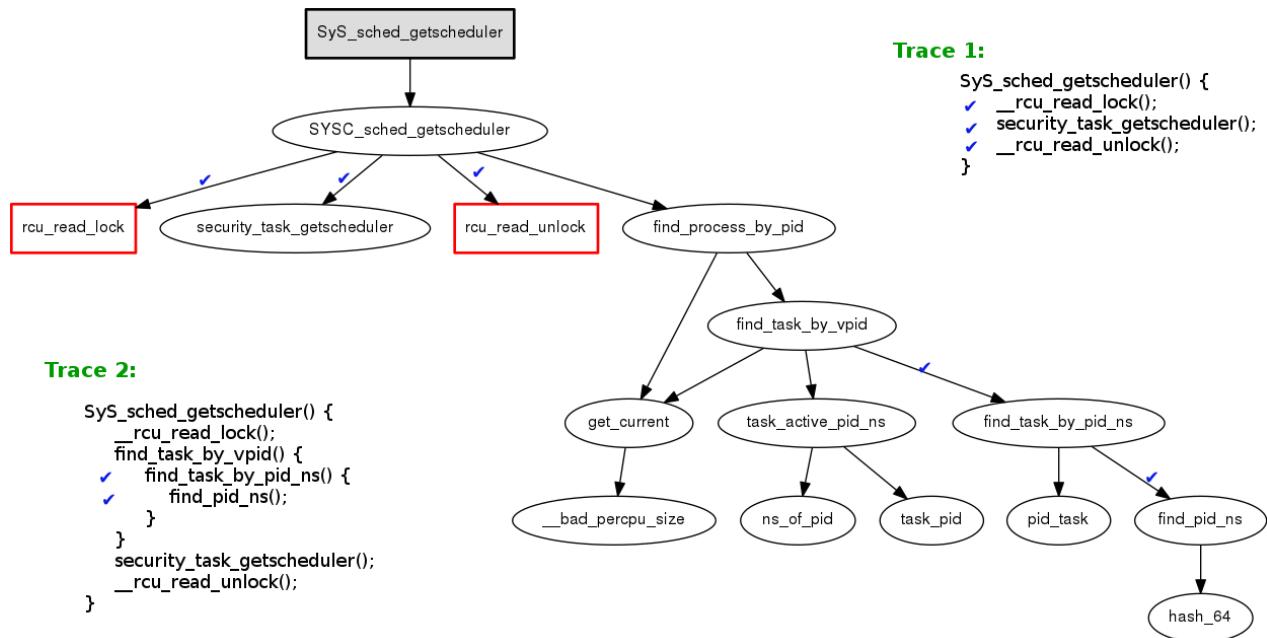


図 3.17 関数コールグラフとトレーサを用いたパスカバレッジ測定例

フォームとして SIL2 対応の目標となっている領域には Linux Kernel に加えて BusyBox と glibc からなる最小構成のユーザ空間も含まれる。そのため、同様の手法をユーザ空間でも適用するべきであると SIL2LinuxMP コミュニティでは認識されており、ユーザ空間における具体的なツールと実現手段は今後の調査検討課題となっている。

関数コールグラフの LOPA への応用

機能安全対応のシステム開発では、必ずしも常にシステム全体に同一の安全水準が求められるわけではない。システム要求や開発リソース・コストなどのマネジメント戦略次第で、特定の部分のみ SIL2 水準の安全性を確保しその他の部分は SIL0(Quality Management (QM) : 安全性の要求なし) 領域とするアーキテクチャ設計も可能である。ただし、このように複数の安全水準を持つサブシステムを混在させる場合は各領域の影響が互いに干渉することのないように対策を施さなければならない。特に SIL0(QM) 非安全領域で欠陥や障害があった場合にその影響が SIL2 安全領域に伝搬することは許容されない。SIL2 安全領域と SIL0(QM) 非安全領域との隔離を実現する手段のひとつとして、SIL2LinuxMP では図 3.18 のように Linux コンテナ技術を利用した構成が検討されている。

GNU/Linux システムにおける一般的なコンテナ技術はユーザ空間から見たリソースを各々のコンテナ内で隔離するものであるが、厳密に kernel 内部のリソースまで隔離して運用することはできない。そのため、本来安全に隔離保護されているべき領域に対するリスク評価を行う Layer of Protection Analysis (LOPA) の実施が求められる。SIL2LinuxMP では DB4SIL2 開発の一環として、関数コールグラフを利用した LOPA 実施方法が次のように検討されている。

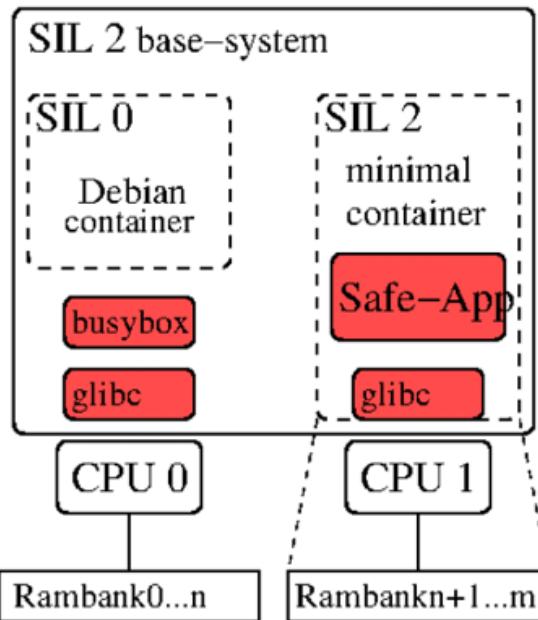


図 3.18 SIL2 安全領域と SIL0(QM) 非安全領域をコンテナで隔離するシステム構成

seccomp と cgroups のリソース独立性を解析する例

Control Groups (cgroups) はプロセスのリソース (CPU、メモリ、ディスク I/O 等) の利用を制限または隔離する Linux Kernel の仮想化機能である。Secure Computing Mode (seccomp) はユーザ空間に対してシステムコールの利用を制限したサンドボックス環境を提供する Linux Kernel のセキュリティ機能である。cgroups と seccomp はともに、ユーザ空間のアプリケーションを隔離することで他のアプリケーションやシステムにその挙動の影響を伝搬させないことを意図して使われるが一般的である。互いにその挙動が独立であることが想定されているならば、各々の機能から使われる関数も呼び出し関係は分離されているはずであると予想される。ここで、cgroups と seccomp の両方の機能を無効にしたときに使われるコード領域に含まれる関数の集合を *BASE*、seccomp だけを有効にしたときの領域に含まれる関数の集合を *SEC*、cgroups だけを有効にしたときの領域に含まれる関数の集合を *CGR* として各々の関係を考える (図 3.19)。

seccomp 機能と cgroups 機能が独立であるならば *SEC* と *CGR* の共通部分は *BASE* の部分集合であるはずであるが、*SEC* と *CGR* の共通部分に属しかつ *BASE* に属さない関数 ($\in (SEC \cap CGR) \setminus BASE$) が存在する (図 3.20)。このような共通部分に属する関数は、各々 seccomp 機能と cgroups 機能に関する関数コールグラフを導出することで調べることが可能である。seccomp と cgroups から共通して使われる関数が特定されれば、それらの共通関数に範囲を限定してレビューや検証のコストを集中させることで各々の機能の独立性を効率的に調査することができる。つまり本項で述べた戦略は機能の挙動が完全に隔離されていることの証明を目指すのではなく、各々から共通して参照される範囲を特定した上でその限定された部分が共通して参照されても問題ないことを示すかまたは必要に応じて何らかの対策を施すという、IEC61508-3 7.4.2.12 Route 3₅: assessment of non-compliant development の原則に基づくものである。

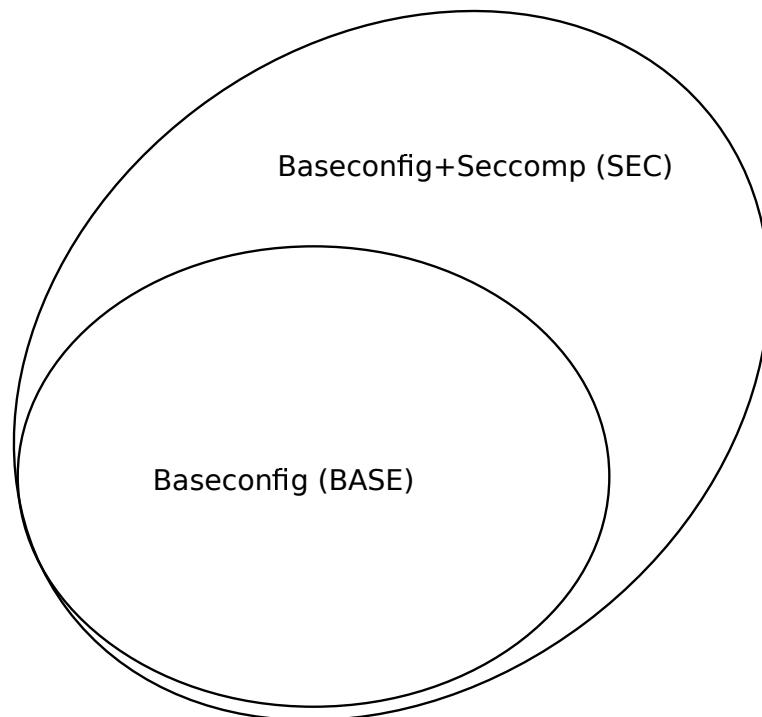


図 3.19 seccomp 機能を有効にしたときに使われる関数の集合 *SEC* と、無効にしたときの関数の集合 *BASE*

seccomp と cgroups の CCF 分析例

ソフトウェアの開発履歴情報を共通原因故障 (CCF) 分析に応用することが SIL2LinuxMP プロジェクトで検討されている。ソースコードの記述スタイルは各プロジェクト毎に方針が存在し統一されていることが多いものの、コードには開発者毎に思考の癖や好みのパターンが表れる。そのような開発者依存のコーディングパターンがあるロジック実装で使われたりある別のパターンと併用されたりした場合に何らかの問題を引き起こす原因となることがあり得る。オープンソースソフトウェアの開発には Git に代表されるバージョン管理システムが使われることが一般的であり、そこにはソースコード変更の単位でその開発者名が記録されている。ここで、例えば seccomp プロジェクトの Git コミット履歴から seccomp プロジェクトに参加している開発者を調べれば、cgroups プロジェクト側で発生したバグの原因となるコードを書いた開発者が seccomp のコードにもコミットした履歴があるかどうかが分かる。この情報を手掛かりにして、cgroups プロジェクトのバグと同じ原因で発生する可能性のあるバグが seccomp プロジェクトにも残存する可能性をさらに分析することができる。

ちなみに seccomp と cgroups の両方のプロジェクトにコミット履歴を残している開発者と各コミット数は表 3.1 の通りである。

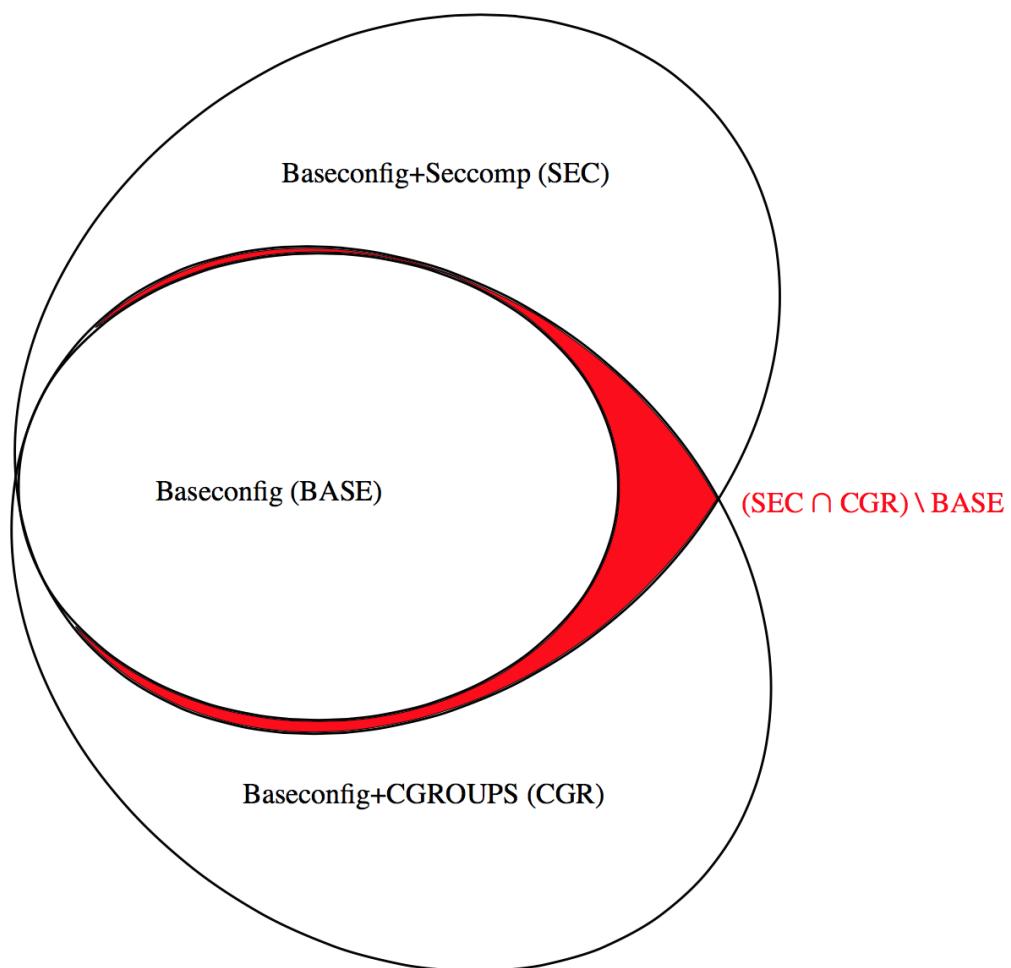
図 3.20 $(SEC \cap CGR) \setminus BASE \neq \emptyset$

表 3.1 seccomp と cgroups に共通する開発者と各コミット数

	seccomp		cgroups
54	Linus Torvalds	2	Linus Torvalds
52	Daniel Borkmann	203	Daniel Borkmann
48	David Howells	3	David Howells
2	Fabian Frederick	1	Fabian Frederick

3.3 質疑や議論を通した SIL2LinuxMP コミュニティへのコントリビューション

本節では、2015年4月からこれまでSIL2LinuxMP活動を通してコミュニティへ働きかけた内容を記載する。SIL2LinuxMPプロジェクトにおける主なコミュニケーション手段はメーリングリストとGitである。プロジェクトに対する活動は各参加者の主体性に任せられている状況であり、全ての活動はメーリングリストおよびGitの履歴に残り最終的にSIL2LinuxMPの成果として参照される。このようなオープンなプロジェクトでは、有益なコントリビューションを多く残している参加者ほどコミュニティに認知される。[2.3](#)節で述べたように、近年はオープンなコミュニティで認知されること自体が長期的に価値のある投資としての意味を持つ。OSSに対する機能安全のノウハウを得るという目的と並んで、日立のSIL2LinuxMPへの貢献を示すため、また筆者自身がOSS活動の進め方を理解するため、研修期間を通してメーリングリストとGitでの活動を活発に行うことを意識した。

3.3.1 メーリングリスト上の活動

[SIL2LinuxMP メーリングリスト](#) [28] はコミュニティでの議論や何らかの告知が行われる公な場である。日立からは、SRSなどのSIL2LinuxMPドキュメントとコンプライアンスルートおよび各種検証手法の不明点の質問をするため、また日立が行った技術調査・検討内容の報告をするためにメーリングリストへの投稿を継続的に行った。表3.2は、メーリングリストへの投稿者別に2015年4月～2016年3月までの投稿数が多い順で投稿者メールアドレスをリストした結果である。日立は4名から投稿があり、そのうち筆者(kotaro.hashimoto.jv@hitachi.com)は36投稿でプロジェクト全参加者中で2番目に投稿数が多かった。1、3番目に投稿数が多いder.herr@hofr.at(Nicholas Mc Guire)とandreas.platschek@opentech.atはSIL2LinuxMPプロジェクトの主催メンバである。これまでこのメーリングリストでは、日立が発した質問に対してNicholasやAndreasが回答しそれに他の参加者が反応して議論が膨らんでいく、という形式で話が進むケースが多かった。

表 3.2 個人別 SIL2LinuxMP メーリングリスト投稿数ランキング (2015 年 4 月 ~ 2016 年 3 月)

投稿数	投稿者メールアドレス
40	der.herr@hofr.at (Nicholas Mc Guire)
36	kotaro.hashimoto.jv@hitachi.com
16	andreas.platschek@opentech.at (andi@opentech.at 含む)
15	Mandar.Pitale@bmw-carit.de
13	julia.lawall@lip6.fr
13	krishnaji@hitachi.co.in
12	C.Emde@osadl.org
8	Lukas.Bulwahn@bmw-carit.de
7	georg.schiesser@opentech.at (george.schiesser@gmail.com 含む)
6	Bernhard.Noelte@tuev-sued.de
3	office@osadl.org
2	Georg.Waibel@sensor-technik.de
2	KaziKhaled.Al-Zahid@bmw-carit.de
2	Tilmann.Ochs@bmw.de
1	Ursula.Braun@osadl.org
1	Tilmann.Ochs@bmw-carit.de
1	masami.hiramatsu.pt@hitachi.com
1	Quirin.Gylstorff@kuka.com
1	dvhart@linux.intel.com
1	khoroshilov@ispras.ru
1	pradyumna@hitachi.co.in
1	Kai-Daniel.Niestroj@sensor-technik.de
1	Christian.Hartmann@kuka.com
1	jeremiah.foster@pelagicore.com
1	Mandar.Pitale@bmw.de

表 3.3 は投稿者のメールアドレス情報を元に企業別で投稿数を集計した結果である。日立は SIL2LinuxMP プロジェクト主催元の OpenTech に続いて 51 投稿で、プロジェクト全参加企業中で 2 番目に投稿数が多かった。

表 3.3 企業別 SIL2LinuxMP メーリングリスト投稿数ランキング (2015 年 4 月～2016 年 3 月)

投稿数	投稿者メールアドレス
63	opentech.at (hofr.at 含む)
51	hitachi.com (hitachi.co.in 含む)
29	bmw.de (bmw-carit.de 含む)
16	osndl.org
13	lip6.fr (INRIA)
6	tuev-sued.de
3	sensor-technik.de
2	kuka.com
1	pelagicore.com
1	linux.intel.com
1	ispras.ru

表 3.4 はメーリングリストでの話題（スレッド）別に各話題における投稿数を集計した結果である。これらのうち、日立が初めに投稿した質問または提案で議論が始まったものは右端に ✓ を付けている。

表 3.4 スレッド別 SIL2LinuxMP メーリングリスト投稿数ランキング (2015 年 4 月 ~ 2016 年 3 月)

投稿数	スレッドタイトル名 (の一部)	日立の投稿で議論が始めたもの
19	Question about SRS and verification strategy	✓
16	Return Oriented Programming (ROP)	
11	Next SIL2LinuxMP meeting in September	
10	How could we qualify a toolchain (T3 off-line	✓
10	SRS question: How will COTS upgrade process be	✓
10	Pruning function call graph	✓
8	Gray box Analysis approach	✓
8	Preliminary safety concept available?	
8	Results of Static analysis on Minimized source code	✓
7	sil2linuxmp	
7	Scheduliing of Application	
7	SRS question: SIL0 isolation	✓
6	Scoping targets for Tracing and Coverage	✓
6	Live DVD build issue	✓
5	Addressing the differences between ISO 26262 and IEC	
5	SIL2Linux Meeting in July 2015 (2015-07-22) at	
5	Minimal kernel source code [SIL2LinuxMP scope]	✓
4	Minimal SIL2LinuxMP configuration	✓
4	Questions about Live-DVD and OGSN regarding	✓
4	Question about purpose of the Prequel	✓
4	Comparison between static call graph and dynamic	✓
3	Route 3s: Detailed Design,	
3	How OSS licenses should be handled with	✓
3	Is the "Device Tree" SIL2LinuxMP target component	✓
2	First CM Report 2015-05-20	
2	SIL2LinuxMP tool needs for Graybox process	✓
2	Welcome Hitachi! (and pull request for d5bcaad0)	
2	Third CM Report 2015-08-15	
2	4th CM Report 2015-09-16	
2	iso from Live DVD	
2	Target kernel version/configuration in SIL2LinuxMP	
2	Initial Test	
2	Second CM Report 2015-07-01	
2	Pull request for branch lbulwahn (commit 5671424	
1	SIL2 Project Plan V1.4 with final list of	
1	Welcome Hochschule Heilbronn!	
1	Reminder: SIL2LinuxMP Progress Meeting on September	
1	Network connectivity to the GIT repository	
1	SIL2Linux MP Milestone Meeting will take place on	
1	[GIT PULL] Please pull dvhart: doc make target	
1	SIL2LinuxMP Milestone Conference (Go/No-Go) on	
1	Agenda of the SIL2LinuxMP Milestone Conference	
1	SIL2LinuxMP Live DVD now live on the Internet	
1	SIL2LinuxMP Meetings Schedule	
1	Shadow monitoring at the OSADL QA Farm	
1	FOSDEM 2016 – safety-critical talk in Brussels?	
1	Web-based command line access to the SIL2LinuxMP	
1	5th CM Report 2016-03-01	
1	Reminder: Review of current SIL2LinuxMP	

表3.4を見ると、実質半数以上の議論が日立からのメーリングリスト投稿を契機で行われていることが分かる。日立の投稿により行われた議論は主に以下に示す話題に関するものであった。

- コンパイラの検証手法（3.1.1項で記載）
- DB4SIL2 の目的と GrayBox テストの定義・手法（3.2.2節で記載）
- 関数コールグラフを導出するツール調査（3.1.4項で記載）
- 関数コールグラフを利用したカバレッジ測定手法（3.2.2項で記載）
- デッドコードを削除し検査対象を限定する Code Minimization 技法（3.4節で記載）
- Code Minimization 技法を応用し関数コールグラフを小さくする手法（3.29項で記載）
- SIL2LinuxMP プラットフォームの検証で使われるツール選定（3.2.1項で記載）
- Git コミット履歴をソフトウェア品質測定に測定する手法（3.1.5項で記載）
- Coccinelle の機能安全プロセスにおける使用方法（3.1.1項で記載）
- 安全領域と非安全領域の独立性分析方法（3.2.2項で記載）
- SIL2LinuxMP 開発環境（Live-DVD）のインストール不具合対策
- OGSN 不具合対策
- デバイスツリーの機能安全対策
- 機能安全認証プロセスと OSS ライセンスチェックの関係
- 機能安全対応開発におけるチーム構成と役割定義
- OSS コンポーネントのアップデートプロセス
- Linux Kernel の最小構成設定の作成方法

SIL2LinuxMP メーリングリスト上の活動を通して、3.1節や3.2節で記載したような一連のノウハウを得ることができたほか、SIL2LinuxMP コミュニティを巻き込んで議論を行うことができた。特に SIL2LinuxMP プロジェクトの参加企業が 20 社以上ある中で、レビューパートナーである日立が率先して質疑応答や意見交換を行ってきたことは SIL2LinuxMP コミュニティに対する貢献として十分なアピール効果があったと考えている。ただし表3.3に示す通り、プロジェクト主催者である OpenTech と日立以外の参加企業の活動形跡がほとんど見えない状況であることが大きな懸念である。プロジェクト 2 年目に入る 2016 年 4 月以降では、OpenTech と日立以外の参加企業も活発かつ自主的にプロジェクトに参加していくことができるような何らかのマネジメント対策が早急に必要である。

3.3.2 Git 上での活動

SIL2LinuxMP プロジェクトはドキュメントやツールの開発トレーサビリティを確保するために Git を構成管理ツールとして使用している。日立は、SIL2LinuxMP プロジェクトで独自に開発されるツールのテストやデバッグに参加することを通して、レビューパートナーとして有益なフィードバックを行うことを目標とした。また、各種検証ツールの調査結果の共有や日立が開発した Code Minimization ツール（3.4項で記載）の公開も Git 上で行った。

SIL2LinuxMP が開発するツールへのフィードバック

SIL2LinuxMP プロジェクトが独自に開発をするツール類には以下のものがある。

Live-DVD： SIL2LinuxMP 活動を行うための統合開発環境。Debian Live で作成される。

OSADL GSN Editor (OGSN)： [3.1.6 項](#)で記載。

DB4SIL2： [3.2.2 項](#)で記載。

Live-DVD は、SIL2LinuxMP で選定または開発されたツールセットが予めセットアップされた統一環境を提供することを目的に作成される。SIL2LinuxMP 開発環境のベースとしては Debian が Nicholas らによるアセスメントにより選定されている。日立は Live-DVD のイメージビルドからシステム起動までをテストし、見つかった手順上の不備について各々対策方法を調査してビルドスクリプトおよび手順の修正をコミットした。OGSN についてもスクリプトと手順がまだ成熟しておらず、インストールが成功しない問題や例外処理が多数不足している等の課題があった。OGSN を初めて導入する開発者が手間取ることのないように、日立は手順およびスクリプトの整備を行った。関連して、OGSN と連携している SRS 等のドキュメントにも L^AT_EX コンパイル上の問題があったためドキュメントの生成スクリプトに対する修正を行った。

検証ツール調査内容の共有

SIL2LinuxMP 計画においてはじめの 1 年間は技術調査フェーズと位置づけられており、各種検証ツールの調査とそれらの機能安全対応開発での使用方法検討が重視されていた。テストカバレッジ測定方法の検討 ([3.2.2 項](#)) において、当初 Nicholas と Andreas ら (OpenTech) は関数コールグラフ生成ツールとして cflow を試用していた。しかし cflow には関数ポインタを介した呼び出し関係を表現できないという欠点や、関数名や変数名の扱いに関して [17 ページ](#)で述べたような不備があった。そこで日立は独自に調査を行い、cflow の欠点を補いかつトレーサと併用してカバレッジ測定に用いるという目的に適切な関数コールグラフ生成ツールとして CodeViz が利用できることを SIL2LinuxMP メーリングリスト上で報告した。またその調査実験結果の詳細報告を Git レポジトリにコミットした。

日立が独自に開発した検証手法やツールの公開

SIL2LinuxMP 活動の一環で日立が独自に開発した技法として、ソースコード中でコンパイル対象となる無効な #ifdef コードブロックを削除することで使用されるコードだけを抽出し検査対象となる範囲を限定する Code Minimization がある ([3.4 項](#)で詳述)。Code Minimization のソースコード、試用手順、応用例、技術検討内容の公開を SIL2LinuxMP プロジェクトの Git レポジトリ上で行った。

また SIL2LinuxMP ドキュメント開発において、複数のドキュメントで共通して用いられる用語を一箇所のファイルでまとめて記載し各々のドキュメントから参照する形で管理するための大規模な T_EX ファイルの修正が行われたことがあった。この作業を促進するため、各 T_EX ファイル中でまだ用語が直

接記載されている箇所を特定して要修正箇所としてハイライトするスクリプトを作成し SIL2LinuxMP Git レポジトリ上で公開した。

SIL2LinuxMP Git レポジトリでの活動結果

SIL2LinuxMP Git レポジトリ上で日立が作成または編集したファイルとその内容は次の通りであった。

- doc/investigation/Analysis_tools/callgraphs/CodeViz/CodeViz_investigation_note.txt
 - … 関数コールグラフ生成ツール CodeViz の調査
- doc/investigation/Analysis_tools/callgraphs/CodeViz/fix_callback_traversing.patch
 - … CodeViz 付属のコールグラフデータ視覚化スクリプトの関数ポインタ周りの不具合修正
- doc/investigation/Analysis_tools/callgraphs/CodeViz/ncc_visualization.txt
 - … CodeViz の ncc により生成したコールグラフデータの表示方法に関する調査
- doc/investigation/Analysis_tools/callgraphs/CodeViz/vfs_read.png
- doc/investigation/Analysis_tools/callgraphs/CodeViz/vfs_write.png
 - … CodeViz で生成した関数コールグラフの例
- src/tools/minimization/minimize.py
 - … 使用されないコードを削除して検証範囲を限定する Code Minimization 技法の実装
- src/tools/minimization/README.md
 - … minimize.py の説明書
- src/tools/minimization/EvaluationNote
 - … BusyBox に minimize.py を適用したときのソースコード複雑度測定実験
- pm/weekreports/khashimoto/March_2016
 - … CodeViz で生成する関数コールグラフと、Code Minimization 適用後に cflow で生成したグラフのサイズ比較調査
- doc/investigation/Analysis_tools/callgraphs/PruningCallGraph.txt
 - … Code Minimization 技法で関数コールグラフを枝刈りする応用例
- pm/weekreports/khashimoto/2015-12-16_LiveDVD_SetUp_Note
 - … Live-DVD ビルド手順の不具合修正調査
- src/tools/live-dvd/build.sh
 - … Live-DVD ビルド手順の不具合修正
- src/investigation/ogsn/ogsn3.py
 - … OGSN の例外処理強化
- src/investigation/ogsn/README
 - … OGSN のインストール・使用手順不備の修正
- pm/weekreports/khashimoto/2015-12-15_OGSN_rmToo_Installation_Note
 - … OGSN および rmToo を併用して SRS ドキュメントを生成するための手順調査
- pm/pm_docs/SRS/Makefile
 - … SRS ドキュメントを生成するための L^AT_EX、rmToo、OGSN 統合手順不具合修正
- share/SIL2Linux_Abbreviations.tex
 - … SIL2LinuxMP 用語集の表記誤り修正
- src/tools/acroscan/acroscan.py
 - … T_EX ファイル中にハードコーディングされている用語を検出して参照形式で書き換えるよう修正を促すツール
- src/tools/acroscan/README
 - … acroscan.py の説明書

上記の変更は全て SIL2LinuxMP Git レポジトリの本線に取り込まれている。ただし、SIL2LinuxMP 構成管理チームは各々のプロジェクト参加者のブランチの内容までレビューしているわけではなく、コミットメッセージの形式的な確認のみを行ってマージを行っている状況である。今後は Change Control Board (CCB) が設置されて、変更内容をきちんとレビューした上でマージ判断を行うプロセスが導入される予定である。

図 3.21 は、2015 年 4 月から 2016 年 3 月までに行われた Git コミットについて寄与率 (Commits(%)) が多い順で開発者をリストした結果である。Git コミットの集計には [GitStats \[16\]](#) を用いた。

List of Authors									
Author	Commits (%)	+ lines	- lines	First commit	Last commit	Age	Active days	# by commits	
Nicholas Mc Guire	858 (50.00%)	254087	13529	2015-03-08	2016-03-13	371 days, 13:46:46	199	1	
Georg Schiesser	450 (26.22%)	18275	63472	2015-03-26	2016-03-16	355 days, 12:18:50	104	2	
Andreas Platschek	291 (16.96%)	116053	53847	2015-08-02	2016-03-14	225 days, 0:19:06	91	3	
Kotaro Hashimoto	35 (2.04%)	1621	327	2015-11-20	2016-03-21	121 days, 14:19:56	21	4	
Siro Mugabi	29 (1.69%)	28119	5040	2015-04-11	2015-08-07	118 days, 5:22:08	13	5	
Astrid Spura	10 (0.58%)	366	176	2015-04-27	2015-08-12	107 days, 0:43:33	3	6	
Lucas Böhm	7 (0.41%)	71	36	2016-01-14	2016-02-03	19 days, 22:05:36	4	7	
Waibel Georg	6 (0.35%)	1296	26	2015-11-18	2015-12-03	15 days, 2:59:05	3	8	
Mandar Pitale	6 (0.35%)	439	0	2015-07-02	2015-09-03	62 days, 23:25:28	5	9	
Bernhard Mathias	4 (0.23%)	86	86	2016-02-09	2016-02-09	0:01:43	1	10	
Darren Hart	3 (0.17%)	176	35	2016-02-27	2016-03-05	7 days, 0:19:07	2	11	
Carsten Emde (Work)	3 (0.17%)	414	355	2015-04-28	2015-06-05	38 days, 11:50:40	3	12	
Carsten Emde	3 (0.17%)	618	565	2015-08-09	2016-02-06	180 days, 20:58:31	3	13	
Ursula Braun	2 (0.12%)	21	16	2015-06-23	2015-12-17	177 days, 5:11:27	2	14	
Robert Bertossi	2 (0.12%)	37	34	2016-03-08	2016-03-15	7 days, 3:56:31	2	15	
Kazi Khaled Al-Zahid	2 (0.12%)	351	0	2015-09-16	2015-09-16	7:44:50	1	16	
Georg Waibel	2 (0.12%)	42	38	2015-06-23	2015-11-18	148 days, 3:48:19	2	17	
siro	1 (0.06%)	149	0	2015-04-06	2015-04-06	0:00:00	1	18	
Lukas Bulwahn	1 (0.06%)	128	128	2015-06-05	2015-06-05	0:00:00	1	19	
Andrea Ruf	1 (0.06%)	2	2	2015-06-25	2015-06-25	0:00:00	1	20	

図 3.21 開発者別 SIL2LinuxMP Git コミット寄与率集計結果

コミット寄与率 Top3 の Nicholas Mc Guire, Georg Schiesser, Andreas Platschek は SIL2LinuxMP を主催する OpenTech のメンバーで、ドキュメント執筆、ツール調査・開発を主導している。筆者 (Kotaro Hashimoto) は 35 コミット (2.04%) で、OpenTech メンバに続き 4 番目に寄与率が大きかった。[3.3.1](#) 項でも述べたとおり、OpenTech と日立以外の参加企業からのコミットが非常に少ないことがプロジェクト上の懸念である。今後 SIL2LinuxMP プロジェクトを盛り上げるために日立としてできる対策を考え、コミュニティが自発的に動き出すような仕掛けをしていくことがプロジェクト成功に不可欠である。

3.3.3 Face to face meeting

SIL2LinuxMP プロジェクトは参加者間のコミュニケーションルートとしてメールングリストと Git の他に face to face 形式のミーティングやワークショップを重視している。2016 年 3 月第二週にはドイツ・ハイデルベルグでプロジェクト参加者全員を対象とするマイルストンミーティングが行われた。こ

こでは SIL2LinuxMP プロジェクトが始まってから約 1 年となるタイミングで、これまでの進捗と成果・残課題の共有および 2 年目の方針についての議論が行われた。この場で日立からは Code Minimization Strategy と題した技術発表を行った。Code Minimization 技法は既存の様々なソフトウェア検証・解析手法のパフォーマンスを向上させる可能性を持ち、機能安全認証プロセスにとっても非常に有効な戦略となり得る技術として SIL2LinuxMP コミュニティから高い評価を受けた。ミーティング終了後には OpenTech と日立とで別途会合を行い、より突っ込んだ質疑応答や意見交換を行った。マイルストンミーティングでは、これまでメールと Git だけを介してコミュニケーションを取ってきた技術者と直接顔を合わせて会話ができ、素朴な疑問や率直な意見をぶつけることでオンラインで行うよりも有意義な議論ができたと感じた。

3.4 Code Minimization 技法の開発と応用例提案

本章では日立が開発した Code Minimization 技法を述べる。Code Minimization は、ソースコード中で使用されないコード（無効な `#ifdef` ブロック）を削除することで検証範囲を限定する技法である。これにより次のような効果が得られる。

- ソースコードの可読性向上により効率的なレビューができるようになる
- 検査対象が最小化されることによりソフトウェア検証に要するコスト（時間・空間計算量）を抑えことができる
- 無駄な検証が省略されることで検証結果の品質が向上する（False-Positive 抑制、カバレッジ向上等）

3.4.1 問題設定

Linux Kernel や BusyBox のソースコードには、様々な環境で動作させるための数多くの設定項目が用意されている。各設定項目に対応する機能はコンパイル時に有効・無効を切り替えるため `#ifdef` や `#if` で括られて実装されることが多い。このようなコンパイルスイッチは適切に用いないと可読性が低下する原因となる。場合によっては、コンパイルスイッチを乱用した結果コードが複雑になりすぎて深刻なソフトウェア品質の低下を招くこともあり得る。図 3.22 は、Linux Kernel のコードでコンパイルスイッチが複雑に絡み合っている例である（`/drivers/dma/dmaengine.c`）。

図 3.22 の `device_has_all_tx_types()` 関数では各設定値 `CONFIG_***` に関する論理式が複数の `#if` または `if` ブロックでネストされており、全ての設定値を覚えていないとどこがコンパイル時に有効になるかを瞬時に把握することが困難である。特に否定形の制御（`#ifndef` や `if(! ...)`）は人間にとつて直感的に理解しづらい傾向があるため、`CONFIG_***_DISABLE_***` や `#ifndef`、`#if !defined`、`if(! ...)` 等が連結しているロジックは誤りの原因となることが多い。このように、コンパイルスイッチは使い方によってはソフトウェアの挙動の理解を妨げ、デバッグやレビューの効率を低下させ、保守性を低下させ、結果として全体の品質低下を招く可能性がある。例えばソフトウェアの挙動をコードと対応させて追いたい場合には、使われていない `#ifdef` ブロックが一掃されたソースがあれば効率の良い解析が実施できると考えられる。我々はこの問題に対して可読性の高いソースコードを得ることを目的に、有効な `#ifdef` のみからなるソースコードを抽出する方法を調査した。OpenTech の Nicholas Mc Guire も同様な問題意識を持っており、2015 年 5 月に [GCC コミュニティに対してコンパイラオプションに関する質問を行っている](#) [1]。

既存手法の調査を行った結果、Stack Overflow 上の記事「[設定ファイルを元に Linux Kernel 中の使われないコードを削除する方法](#)」[29] で手掛かりを得た。この記事で提案されている手順の概要は次のようなものであった。

```
static bool device_has_all_tx_types(struct dma_device *device)
{
    /* A device that satisfies this test has channels that will never cause
     * an async_tx channel switch event as all possible operation types can
     * be handled.
    */
#ifndef CONFIG_ASYNC_TX_DMA
    if (!dma_has_cap(DMA_INTERRUPT, device->cap_mask))
        return false;
#endif

#ifndef CONFIG_ASYNC_MEMCPY || defined(CONFIG_ASYNC_MEMCPY_MODULE)
    if (!dma_has_cap(DMA_MEMCPY, device->cap_mask))
        return false;
#endif

#ifndef CONFIG_ASYNC_XOR || defined(CONFIG_ASYNC_XOR_MODULE)
    if (!dma_has_cap(DMA_XOR, device->cap_mask))
        return false;
#endif

#ifndef CONFIG_ASYNC_TX_DISABLE_XOR_VAL_DMA
    if (!dma_has_cap(DMA_XOR_VAL, device->cap_mask))
        return false;
#endif
#ifndef CONFIG_ASYNC_TX_DISABLE_PQ_VAL_DMA
    if (!dma_has_cap(DMA_PQ_VAL, device->cap_mask))
        return false;
#endif

    return true;
}
```

図 3.22 Linux Kernel /drivers/dma/dmaengine.c の一部

1. ビルドログを取得して全ての実行された gcc コマンドを抽出する
2. 各 gcc コマンドからコンパイルオプション -c 以降を消して -E -fdirectives-only を加える
3. 手順 2 で作成した gcc コマンドを実行する

-E は #include、#define、ifdef 等に対するプリプロセスのみを行う gcc オプションで、-fdirectives-only はプリプロセッサディレクティブへの処理のうちマクロの展開を抑制するオプションである。Stack Overflow の記事では上記のように gcc オプションを駆使することで有効コードのみの抽出が可能であることが示されていたが、この方法を Linux Kernel ツリー全体に対して一括して適用する方法が未解決であった。

我々はソースツリー全体でコンパイル対象のコードを抽出する方法を開発することを目標に、問題を次のように設定した。

Code Minimization の定義

Code Minimization は以下の条件を満たすソースツリーを生成するソースコード変換処理とする。

- コンパイル対象外の `#ifdef` や `#if` ブロックを含まない。
- `#include` 行や `#define` 行は元の形のまま保持される。
- ビルド対象のソースファイルのみを含む。
- 素のソースツリーでビルドした結果と同じバイナリを生成する。

ここで”minimization”という言葉は”minimal configuration”とは異なる意味で用いている。ソフトウェアに関して”minimization”と言う場合一般的には「最小構成の設定」を作ることを意味するが多いが、ここでは上記の Code Minimization の定義に基づくソースコードの変換処理を意味するものとする。

3.4.2 実現方針

Code Minimization を実現するために、Linux Kernel を対象として以下の方針で実装の検討を行った。

- (i) 既存 `Makefile` の仕組み上で動作する構成とする
- (ii) プリプロセスコマンドを構築・実行する
- (iii) 展開されたヘッダファイルの中身と余分な空白行を削除する

(i) 既存 `Makefile` の拡張

Linux Kernel の `Makefile` には、コンパイルと並行して何らかのソースコードチェックツールを実行するための `CHECK` 変数が用意されている。既定では `sparse` が設定されており、これは `make` コマンドで `CHECK` 変数を上書きすることで任意のチェックツールに置き換えることができる。`CHECK` 変数で指定されたチェックツールは `make` コマンドで `C` フラグを設定することにより実行される（図 3.23）。

```
kotaro@kotaro-OptiPlex-7020:~/Minimization/linux-4.3.3$ make help | grep CHECK
make C=1      [targets] Check all c source with $CHECK (sparse by default)
make C=2      [targets] Force check of all c source with $CHECK
```

図 3.23 Linux Kernel の `Makefile` で利用可能なソースコードチェック機能

`CHECK` 変数で指定したチェックツールに対するオプションは、同じく `make` コマンド中の `CF` 変数で与えることができる（図 3.24）。

Code Minimization の実装では、Linux Kernel の `Makefile` に既に存在する図 3.23 と図 3.24 の機能

```
CHECK      = sparse
CHECKFLAGS := -D__linux__ -Dlinux -D_STDC__ -Dunix -D__unix__ \
              -Wbitwise -Wno-return-void $(CF)
```

図 3.24 Linux Kernel の Makefile 中で設定されているソースコードチェックツールとそのオプション

を利用して下記のようなコマンドで Code Minimization を実現する方針とした。

Code Minimization コマンド形式

```
make C=1 CHECK=minimize.py CF="-mindir ..//minimized-tree"
```

`minimize.py` は Code Minimization の Python スクリプトによる実装である。CF 変数で変換後のソースツリー生成場所を指定する。これにより、通常の kernel ビルドと同時に各々の C ソースファイルに対する Code Minimization が一括して実行される。CHECK 変数で指定したスクリプト `minimize.py` には、各々の C ソースファイルに対する `gcc` コマンドと同じオプション文字列が自動的に渡される。そのため、CHECK 変数を利用する方法は前述の Stack Overflow の手順で必要だったビルドログの取得や `gcc` コマンドの編集といった中間処理が不要であり、全ての処理を 1 コマンドで完結できる利点がある。`minimize.py` スクリプト本体およびその詳しい使用手順は GitHub の [minimization レポジトリ](#) で公開している [9]。

(ii) プリプロセスコマンドの構築と実行

以降は `minimize.py` スクリプトの実装について処理の概要を記載する。

`minimize.py` は `make` プロセスから呼ばれるとき、そのターゲット C ファイルへの `gcc` コマンドと同じオプション文字列を引数で受け取る。`minimize.py` は受け取ったオプション文字列の先頭に `gcc -E -fdirectives-only` を追加した上でその文字列をシステムコマンドとして実行する。その結果、`#ifdef` が消え、`#include` が展開され、`#define` がそのまま保存されたプリプロセス出力が得られる。

(iii) 展開されたヘッダファイルの中身と余分な空白行の削除

`minimize.py` はプリプロセスの結果に対して `#include` で展開されたヘッダファイルに該当する箇所を削除する。`gcc` のプリプロセス出力には以下のようない形式で `#include` の展開情報が残されており、これらは *linemarkers* と呼ばれる。プリプロセス出力からヘッダファイル展開箇所を特定するためには、*linemarkers* を手掛かりとして用いる。

- プリプロセス出力中の *linemarkers* の例

```
# 30 "/usr/include/sys/utsname.h" 2
```

この *linemarkers* の例は「この行以降のコードは /usr/include/sys/utsname.h の 30 行目が展開されたもので、かつ別の入れ子になった #include の展開内容から返ってきた直後」であることを意味している。*linemarkers* の形式については以下のページが詳しい。

[Preprocessor Output - The C Preprocessor \[25\]](#)

minimize.py は、*linemarkers* を解析することで元の C ファイルに存在しないヘッダファイルの展開内容を削除する。その上で元の C ファイルの内容と比較をして #include 行の復元と、消えた #ifdef ブロックに該当する空白行の削除を行う。元のソースと最終的に得られるソースとの差分は無効な #ifdef ブロックの削除のみであり、#include 行および #define 行は元のまま保持される（図 3.25）。

```
C:\Users\khashimoto\Desktop\hoger\uname.c
131 if (toprint == 0) ( /* no opts => -s (sysname) */
132     toprint = 1;
133 )
134
135 uname(&uname_info.name); /* never fails */
136
137 #if defined(__sparc) && defined(__linux__)
138 if (fake_sparc && (fake_sparc[0] | 0x20) == 'y') {
139     strcpy(uname_info.name.machine, "sparc");
140 }
141#endif
142 strcpy(uname_info.processor, unknown_str);
143 strcpy(uname_info.platform, unknown_str);
144 strcpy(uname_info.os, CONFIG_UNAME_OSNAME);
145#endif
146 /* Fedora does something like this */
147 strcpy(uname_info.processor, uname_info.name.machine);
148 strcpy(uname_info.platform, uname_info.name.machine);
149 if (uname_info.platform[0] == 'i'
150     && uname_info.platform[1]
151     && uname_info.platform[2] == '8'
152     && uname_info.platform[3] == '6'
153 ) {
154     uname_info.platform[1] = '3';
155 }
156#endif
157
158 delta = utsname_offset;
159 fmt = "%s" + 1;
```



```
C:\Users\khashimoto\Desktop\hoger\uname.c.minimized
126 if (toprint == 0) ( /* no opts => -s (sysname) */
127     toprint = 1;
128 )
129
130 uname(&uname_info.name); /* never fails */
131
132 strcpy(uname_info.processor, unknown_str);
133 strcpy(uname_info.platform, unknown_str);
134 strcpy(uname_info.os, CONFIG_UNAME_OSNAME);
135
136 delta = utsname_offset;
137 fmt = "%s" + 1;
```

図 3.25 Code Minimization の実行結果例

3.4.3 実行結果

minimize.py は当初 Linux Kernel に対して適用することを想定して実装されたが、Makefile 中の CHECK 変数を利用する仕組みがそのまま BusyBox にも適用できることが分かった。Linux Kernel および BusyBox に対して *minimize.py* を実行した結果削除されたソースコード量を表 3.5 に示す。ここでは実験対象として Linux Kernel 4.3.3 と BusyBox 1.24.15 を使用し、各々 allnoconfig と defconfig を適用したときの Code Minimization 結果を記載している。実験は Ubuntu 14.04 (x86_64) で実施した。

表 3.5 Linux Kernel と BusyBox への Code Minimization 適用で削除されたコード量

設定 \ ターゲット	Linux Kernel 4.3.3	BusyBox 1.24.15
defconfig (x86)	103144 行 (全体の 5%)	20453 行 (全体の 11%)
allnoconfig	64684 行 (全体の 22%)	5945 行 (全体の 34%)

Linux Kernel、BusyBox ともに allnoconfig 設定のときにコード削減割合が高い。これは、無効化される機能が多くなるほど使用されない #ifdef ブロックが増えることと対応している。

`minimize.py` 実行の結果生成されたソースツリーは素のソースツリーと同じ手順でビルドが可能であった。ビルドの確認は、Code Minimization 後のファイルとディレクトリをそのまま素のソースツリー上に上書きコピーして実施した。これは Code Minimization 生成物に `Makefile` などのビルドに必要なスクリプト類が含まれないためである。

Code Minimization は無効コードを削除するのみで、機能そのものには一切影響を与えてはいけない。このことを確認するため、素のソースツリーのビルド生成物と Code Minimization で生成されたソースツリーのビルド生成物の比較を行うことを検討した。各々ビルドの結果生成されたバイナリを `objdump -d` コマンドで逆アセンブリしたところ、アセンブリコードの比較結果は表 3.6 のようになつた。

表 3.6 Code Minimization 後のソースツリーによるビルド生成物と素のソースツリーによるビルド生成物とのアセンブリコード比較結果

設定 \ ターゲットファイル	vmlinux.o	busybox (実行ファイル)
<code>defconfig (x86)</code>	差分あり	一致
<code>allnoconfig</code>	一致	一致

アセンブリコード (`objdump -d` の出力) が一致するターゲットと設定の組み合わせに対してもバイナリファイルの直接比較結果は一致しない。これはビルド生成物にビルドタイムスタンプ等の情報が埋め込まれていることが要因の一つである。`objdump -d` は実行ファイル中から実行部 (executable section) のアセンブリコードを出力する。表 3.6において `objdump -d` が一致することは、`minimize.py` の処理内容が最終ビルド生成物の挙動に影響を与えないことを支持する。しかし、Linux Kernel を `defconfig (x86)` で Code Minimization 適用した場合は、ビルド生成物のアセンブリコードと素のソースツリー由来のものとで差分があった。この事象に対する調査、および `minimize.py` がソフトウェアの機能そのものに一切影響を及ぼさないことの更なる検証方法検討は今後の課題である。

Linux Kernel および BusyBox に対する Code Minimization 実装は以下の GitHub レポジトリで公開している（図 3.26）。[9]

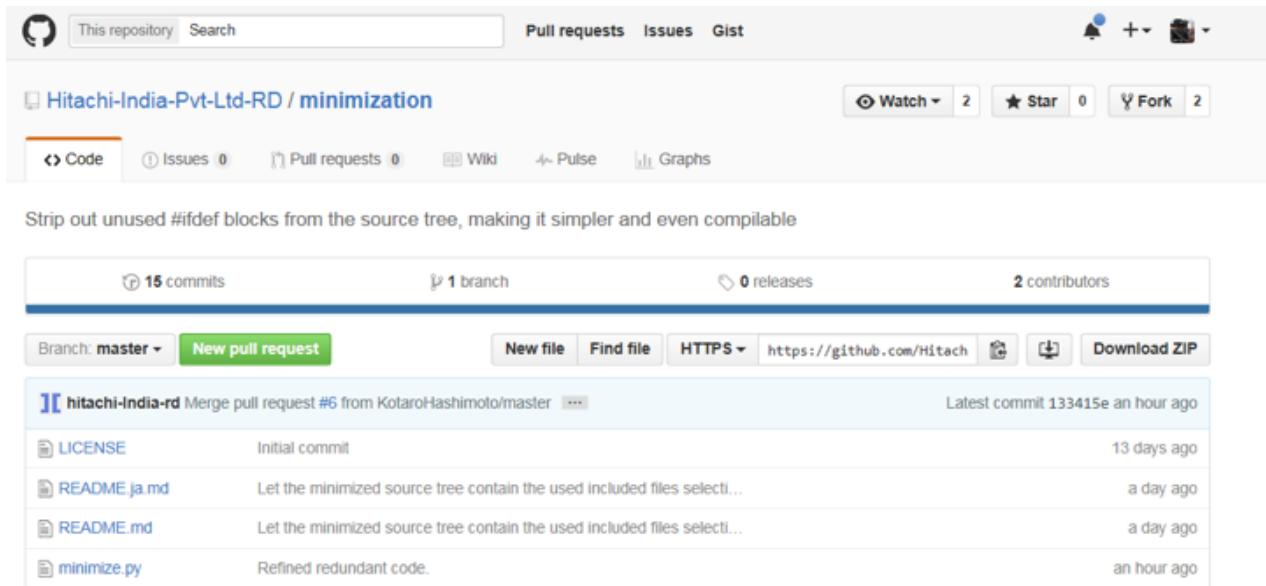


図 3.26 <https://github.com/Hitachi-India-Pvt-Ltd-RD/minimization>

3.4.4 Code Minimization 技法の利点と応用

Code Minimization 技法を適用することで、使用されないコードを削除することによるソースコードの可読性向上が期待される。加えて、ソースコードのレビューや静的検証を行う際の検査対象範囲が限定される効果も考えられる。本項では、可読性向上に関する評価と、検査対象範囲を限定することにより静的検証のパフォーマンスを向上させる応用例について記載する。

変換後のソースコード複雑度評価

Code Minimization 適用によるソースコード可読性向上度合いを評価するため、ソースコードの複雑度を測定するツール [Complexity](#) [12] を使用した。Complexity は C プログラムの複雑度を定量的に測定するために設計された GNU ツールである。コード複雑度の測定ツールとしては例えば McCabe が知られている。McCabe は主に実行パス数に基づく情報からテストに必要な労力を数値化することを目的としているのに対して、Complexity はコードの長さや条件分岐の複雑さを評価することで、特に人間がコードを理解するための労力を数値化することを意図している。

以下は、Code Minimization 適用前の Linux Kernel ソースツリー中の C ソースファイルについて Complexity で複雑度を測定した結果である。”Complexity Histogram”では、関数ごとに測定された複雑度スコアの分布 (0 ~ 1999) がコード行数の積算を度数として示されている。複雑度スコアの平均値 (Average line score) は 23、中央値 (50%-ile score) は 4、最大値 (Highest score) は 1846 であった。

素の Linux Kernel ソースツリーに対する Complexity 測定結果

```
$ find linux-4.4.1 -name "*.c" | xargs complexity -h
Complexity Histogram
Score-Range Lin-Ct
 0-9      277794 *****
 10-19     49923 *****
 20-29     17566 ****
 30-39     7189 **
 40-49     2148
 50-59     1961
 60-69     630
 70-79     563
 80-89     633
 90-99     1381

100-199    5332 *
200-299    2765 *
300-399    0
400-499    0
500-599    0
600-699    0
700-799    0
800-899    0
900-999    0

1000-1999   2345 *

Scored procedure ct:      18176
Non-comment line ct:      370230
Average line score:       23
25%-ile score:           2 (75% in higher score procs)
50%-ile score:           4 (half in higher score procs)
75%-ile score:           9 (25% in higher score procs)
Highest score:            1846 (setgamma() in linux-4.4.1/drivers/media/usb/gspca/topro.c)
```

次に、`allnoconfig` 設定で Code Minimization 適用した Linux Kernel ソースツリーに対して複雑度を測定した結果を示す。複雑度スコアの平均値 (Average line score) は 5、中央値 (50%-ile score) は 2、最大値 (Highest score) は 158 であった。なお、Code Minimization 適用前の測定結果で複雑度スコアが最大であった関数 `setgamma()` (`linux-4.4.1/drivers/media/usb/gspca/topro.c`) は Code Minimization 適用後のソースツリーに含まれていなかった。

allnoconfig 設定の Linux Kernel ソースツリーに対する Complexity 測定結果

```
$ find minimized-tree -name "*.c" | xargs complexity -h
Complexity Histogram
Score-Range Lin-Ct
 0-9      85190 ****
 10-19     7640 ****
 20-29     1004 *
 30-39     944 *
 40-49      102
 50-59      109
 60-69      96
 70-79       0
 80-89       0
 90-99       0

 100-199    396

Scored procedure ct:      7870
Non-comment line ct:    95481
Average line score:      5
25%-ile score:          1 (75% in higher score procs)
50%-ile score:          2 (half in higher score procs)
75%-ile score:          5 (25% in higher score procs)
Highest score:           158 (zlib_inflate() in minikern/lib/zlib_inflate/inflate.c)
Unscored procedures:     4
```

続いて、`defconfig (x86)` 設定で Code Minimization 適用した Linux Kernel ソースツリーに対して複雑度を測定した結果を示す。複雑度スコアの平均値 (Average line score) は 7、中央値 (50%-ile score) は 3、最大値 (Highest score) は 194 であった。

defconfig(x86) 設定の Linux Kernel ソースツリーに対する Complexity 測定結果

```
$ find minimized-tree -name "*.c" | xargs complexity -h
Complexity Histogram
Score-Range Lin-Ct
 0-9      675258 ****
 10-19    86562 ***
 20-29    27818 **
 30-39    12037 *
 40-49     5734 *
 50-59     2583
 60-69     3124
 70-79     3047
 80-89     281
 90-99     1051

 100-199   2135

Scored procedure ct:      50155
Non-comment line ct:    819630
Average line score:       7
25%-ile score:           2 (75% in higher score procs)
50%-ile score:           3 (half in higher score procs)
75%-ile score:           7 (25% in higher score procs)
Highest score:            194 (inflate_fast() in minikern/lib/zlib_inflate/inffast.c)
Unscored procedures:       9
```

以上を総合すると、複雑度スコアの平均値、中央値、最大値はいずれも `allnoconfig < defconfig < (Code Minimization 非適用)` の関係となり、Code Minimization によって削除されたコード量の大小と対応する結果が得られた（表 3.7）。

表 3.7 Linux Kernel ソースツリーの Code Minimization 適用前後に対する Complexity スコア測定結果

スコア種類 \ 設定条件	<code>allnoconfig</code>	<code>defconfig (x86)</code>	Code Minimization 非適用
平均値 (Average line score)	5	7	23
(25%-ile score)	1	2	2
中央値 (50%-ile score)	2	3	4
(75%-ile score)	5	7	9
最大値 (Highest score)	158	194	1846

同様の複雑度測定評価を BusyBox に対して行った結果を表 3.8 に示す。BusyBox での測定結果においても Linux Kernel と同様に概ね `allnoconfig < defconfig < (Code Minimization 非適用)` の関係が得られた。

表 3.8 BusyBox ソースツリーの Code Minimization 適用前後に対する Complexity スコア測定結果

スコア種類 \ 設定条件	allnoconfig	defconfig (x86)	Code Minimization 非適用
平均値 (Average line score)	19	21	22
(25%-ile score)	2	3	3
中央値 (50%-ile score)	5	9	9
(75%-ile score)	21	24	25
最大値 (Highest score)	283	283	283

以上から、Code Minimization で無効コードを削除した結果では Complexity ツールによる客観的な指標においてソースコードの可読性が向上することが示された。

検査対象範囲の最小化

本項では、Code Minimization がソースコードの検査対象範囲を限定する効果に着目し、その性質を応用することで静的検証におけるパフォーマンスを向上させることができる例を示す。

テストカバレッジ測定の原理を考えたとき、一般にテストカバレッジはテスト実施した行数 ($LOC_{exercised}$) の総コード行数 (LOC_{total}) に対する比で表される（式 (3.2)）。

$$TestCoverage = \frac{LOC_{exercised}}{LOC_{total}} \quad (3.2)$$

ここで、コンパイル対象とならない無効な `#ifdef` コードブロックがソースコードに含まれる場合は無効なコードブロック行数 LOC_{unused} を LOC_{total} から引くことでより正確なカバレッジが測定できる（式 (3.3)）。

$$TestCoverage = \frac{LOC_{exercised}}{LOC_{total} - LOC_{unused}} \quad (3.3)$$

式 (3.3) は、使用されないコードを多く特定するほど計算上テストカバレッジが向上する（より正確なカバレッジ評価となる）ことを意味する。使用されないコードが特定されない限り、 $LOC_{exercised}$ を増やすためにどれだけテストに労力を費やしたとしても達成できるカバレッジは 100% 未満 ($1 - \frac{LOC_{unused}}{LOC_{total}}$) で制限される（図 3.27）。ここで Code Minimization 適用により LOC_{unused} を明らかにすれば、達成し得るテストカバレッジをその分向上させることができる。このように、Code Minimization は検査範囲を最小化する効果がある。この技法を”minimization”と呼ぶことはこの意味での「最小化」の効果が得られることに由来する。

関数コールグラフを枝刈りする応用例

Code Minimization で予めソースコードを最小化することで関数コールグラフの解析が簡単になる例を示す。3.2.2 項で述べたように、関数コールグラフはトレーサと組合わせてパスカバレッジを測定する目的で使用できる。関数コールグラフ導出により実行され得る全関数呼び出しパスを特定することは、式 (3.3) において分母 ($LOC_{total} - LOC_{unused}$) に相当にする箇所を測定することに対応する。

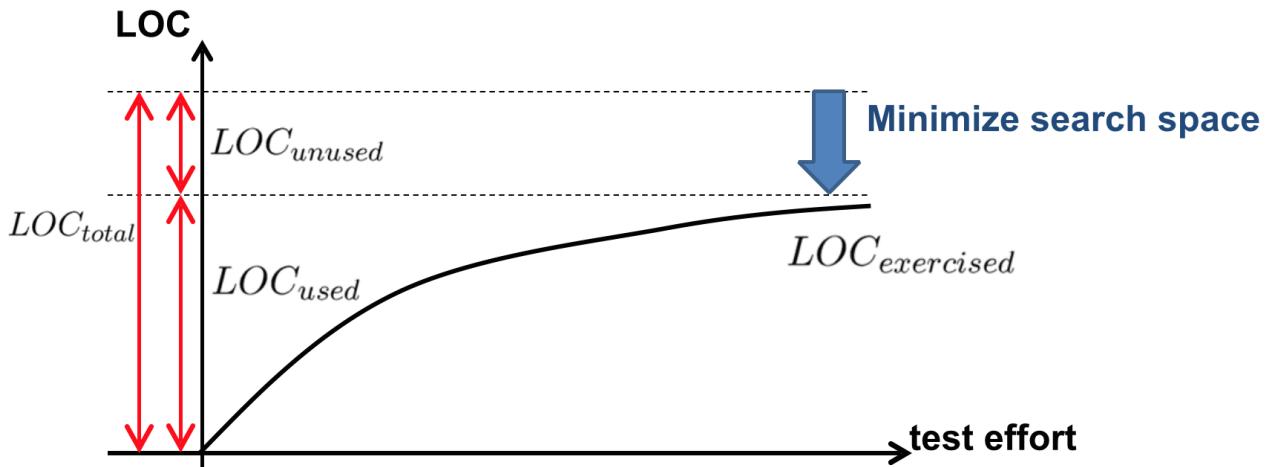


図 3.27 Code Minimization 適用により正確なテストカバレッジ測定が可能となる

ここでは cflow を使用して BusyBox の `init_main()` (`init/init.c`) に対する関数コールグラフを生成する。グラフ画像生成は次のコマンドで実施した。

- cflow で BusyBox から `init_main()` の関数コールグラフを生成する手順

```
$ cd busybox-1.24.1
$ cflow -d 5 -b init/init.c | tree2dotx | dot -Tsvg -o init.svg
```

Code Minimization 適用前の関数コールグラフを図 3.28 に示す。このグラフは `#ifdef` ブロックで無効化される関数も含まれているため複雑で非常に見通しが悪い。Code Minimization 適用前グラフのノード数は 94、エッジ数は 140 であった。図 3.29 は Code Minimization 適用後の関数コールグラフである。こちらのグラフはコンパイル対象である関数のみが含まれるため複雑さが改善している。Code Minimization 適用後のグラフのノード数は 85、エッジ数は 123 であった。

以上の結果から、関数コールグラフ解析において事前に Code Minimization を実施することでグラフのサイズを最小化することができ、解析の効率向上やカバレッジ向上等の効果を期待できることが分かる。ちなみに cflow コマンド自体にもプリプロセスオプションが備わっているため、Code Minimization を使用しなくともコンパイル対象外の関数をグラフに含めないことは可能である。ただし、cflow コマンドでプリプロセスを行う場合は gcc コンパイルオプション文字列をビルドログから取得して cflow の `--cpp` オプションで渡す作業を各 C ファイルについて行わなければならず、手間がかかる上に誤りが発生しやすい。それに対して `minimize.py` は 1 コマンドで一括して全ての C ファイルに対するプリプロセスを行うことができる利点がある。

cflow に限らず、Coccinelle や CPAChecker 等の各種検証ツールもそれぞれプリプロセスオプションが備わっていることが多い。ただし、各検証ツールはプリプロセス自体が主目的ではないためそれらのプリプロセス処理は gcc のものと比べて完全でないことが多い、中には独自のプリプロセス挙動を示すものもある。`minimize.py` は変換後のソースツリーも元と同じ手順でビルド可能であることを保証するものであり、これは Code Minimization 技法が特定の検証ツールに依存せず一般に様々な検証技法の前処理

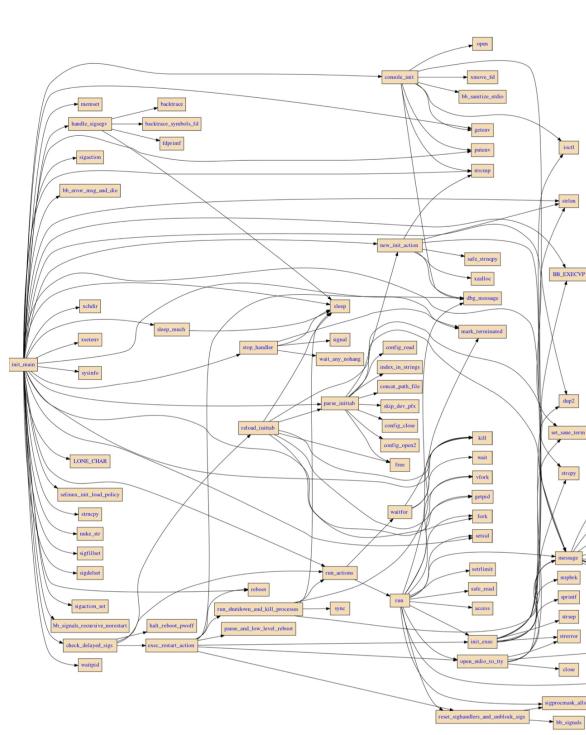


図 3.28 適用前 (94 ノード、140 エッジ)

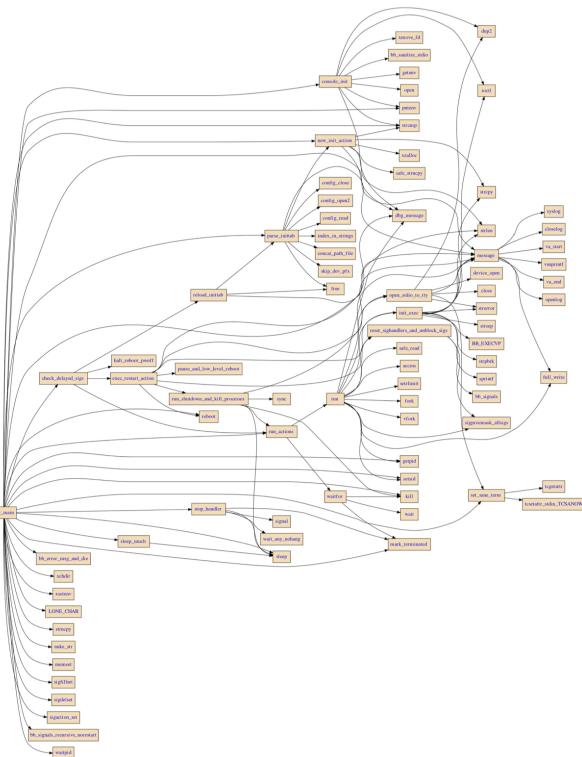


図 3.29 適用後 (85 ノード、123 エッジ)

図 3.30 Code Minimization 適用前後の関数コールグラフサイズ比較

として応用できることを示唆する。

部分ソースツリーの抽出

`minimize.py` は `Makefile` がサポートするサブターゲットに対しても作用する。`minimize.py` を実行する `make` コマンドでサブターゲットを指定した場合、そのサブターゲットが依存するソースファイルのみを含む部分ツリーが抽出される。抽出された部分ツリー中の C ソースファイルは Code Minimization 適用後の状態となる。以下は、BusyBox の `init` サブターゲットから依存されるソースファイルを、コンパイル対象のコードブロックのみを含む形で部分的に抽出するコマンド例である。

- BusyBox の init サブターゲットで使われるソースファイルとコードのみを抽出するコマンド例

```
$ cd busybox-1.24.1
```

```
$ make init C=2 CHECK=minimize.py CF="-mindir ../min-init"
```

上記コマンドの実行結果抽出された部分ソースツリーは図 3.31 のようになる。

指定したサブターゲットからどのファイルが実際にコンパイル対象として使われているかが明らかになることで、例えばコードレビューの効率が向上する効果が期待される。

```
min-init/
└── applets
    └── applets.c
└── include
    ├── applet_metadata.h
    ├── autoconf.h
    ├── busybox.h
    ├── grp_.h
    ├── libbb.h
    ├── platform.h
    ├── pwd_.h
    ├── shadow_.h
    └── xatonum.h
└── init
    ├── bootchartd.c
    ├── halt.c
    ├── init.c
    ├── mesg.c
    └── reboot.h
```

図 3.31 BusyBox の init サブターゲット対象に Code Minimization 適用して得られた部分ツリー

3.4.5 まとめと今後の課題

Code Minimization 技法はソースコードの可読性を向上させる効果がある。この性質はコードレビューやデバッグ効率向上のために利用できる。また、Code Minimization 技法は検査対象範囲を限定する効果がある。これは図 3.14 (IEC61508-3 Table B.2 - Dynamic analysis and testing)において各種カバレッジ 100% を達成するために有力な手段となる。または、テストを実施しない箇所がコンパイル対象外であることに示すエビデンスとして Code Minimization の結果を利用することも考えられる。使用されないコードを特定し検査対象から除外することは、検証ツールが生成する False-Positive 絶対量を抑制する結果にもつながる。さらに、全体のコード量が削減されることで静的検証に要する時間と計算資源コストを節約できる。Nicholas Mc Guire(OpenTech) の実験によると、3 時間要していた Coccinelle 実行が Code Minimization 適用後のソースツリーでは 15 分程度に短縮されたという。

現状 `minimize.py` がサポートするターゲットは Linux Kernel と BusyBox で、変換後ソースツリーのビルドを確認した設定は `allnoconfig`、`defconfig` (x86)、および `omap2plus_defconfig` (ARM ターゲットへのクロスピルド) である。今後はさらに適用できるターゲットと設定を拡充していく予定である。また Code Minimization 適用後のソースツリーが素のソースツリーと機能的に等価であることの検証が未解決課題である。

Code Minimization は一般に検証ツール・手法のパフォーマンスを増強させる技法であるため、組み合わせによって様々な使い方の可能性がある。今後多数の Code Minimization 技法の応用方法を考案し、SIL2LinuxMP コミュニティに提案を行っていく方針である。

第4章

今後の業務への貢献・活用

SIL2LinuxMP プロジェクトは 2017 年 11 月までに完結することが目標とされている。プロジェクト 2 年目となる 2016 年度では、日立はこれまでと同様に SIL2LinuxMP コミュニティへの働きかけを継続するとともに、SIL2LinuxMP プロジェクトで得た知見を社内の製品開発に還元していく活動を行う。2016 年度に実施すべき課題として以下の 3 つを設定する。

- OSS・Linux システムの機能安全対応パイロット開発実施
- 機能安全対応 Linux ディストリビューションの構築と提案
- SIL2LinuxMP コミュニティへの継続的なコントリビューション

4.1 OSS・Linux システムの機能安全対応パイロット開発実施

SIL2LinuxMP プロジェクトは特定のシステムに対して機能安全認証を与えることを目指すのではなく、一般に OSS・Linux を使ったシステムで機能安全認証を達成するためのプロセスおよび方法論を確立することを意図している。そのため、開発されたプロセスと方法論が現実の製品に即して適用できることの事例を残すことが、将来 SIL2LinuxMP の成果によって OSS・Linux が Safety Critical なシステムで使われるようになるために重要である。特に、現在 SIL2LinuxMP で開発されている Compliance Route (IEC61508 文書を Route3_S: assessment of non-compliant development に基づいて OSS・Linux システムに適用できるよう解釈し具体的な手段を対応させたもの) の詳細や実現方法が不完全であると TUeV Rheinland から指摘されている。この問題の一因として、SIL2LinuxMP プロジェクトにおいてターゲットとなるユースケースとシステム構成が定まっていないために、具体的に何に対してどのようにプロセス・方法論を適用するのかがコミュニティ内で明確に共有されていないことが挙げられる。

日立はグループ内で開発されている Linux ベースの Advanced Driving Assistant System (ADAS) に機能安全認証を与えることを目標に SIL2LinuxMP で開発されたプロセスと方法論の適用を試みることで、Compliance Route の具体化と OSS・Linux の SIL2 対応事例確立を目指す。この活動はソースコード自体の開発よりも、開発プロセスのトレーサビリティと開発体制の健全性を確保すること、および定められたプロセスが適切に実施されていることのエビデンスを体系的に蓄積することが中心となる。特に、

使用するソフトウェアの選定、アーキテクチャ設計、リスク分析など特定の個人の評価に依存しがちな様々な判断事項について、その判断に至るまでの議論の経過と判断基準を全て記録することが必要となる。これらの記録は第三者認証機関のレビューを受ける場において、自ら構築したシステムが正しく設計されていることを証明するための材料となる。日立ではこのことを「基本と正道」と言う。同様の概念がサイボウズでは「公明正大」と言われている。[\[33\]](#)

具体的には、SIL2LinuxMP プロジェクトで定期的に行われている Hazard and Operability Studies (HAZOP) セッションを利用して日立 ADAS システムのハザード分析を行う。HAZOP はシステムの潜在危険因子を定量的かつ網羅的に抽出・評価するプロセスで、予め定められた役割と手法によるチームワークで進行する。SIL2LinuxMP コミュニティは IRC と web ベースのツールを活用して HAZOP をオンラインで行う仕組みを構築しており、これによって遠隔メンバの参加や議論内容の確実な保存と事後参照が可能となっている。また DB4SIL2 ([3.2.2 項](#)) を利用して検証結果のエビデンスを保存することや、ソフトウェア部品とツールの選定評価基準を定めた SIL2LinuxMP ドキュメント Acceptance and Test Criteria (ATC) に基づいた評価、および OGSN を利用してその評価過程を可視化する活動を行う。

4.2 機能安全対応 Linux ディストリビューションの構築と提案

SIL2LinuxMP と関連して、日立製品の共通ソフトウェア基盤として使用されることを目的とした独自の Linux ディストリビューション開発が進行している。これは特に Safety Critical な産業・インフラ分野で 10 年単位の長期間にわたって稼動するシステムを意図したものであり、将来的に SIL2LinuxMP の成果を取り込むことで機能安全案件にも対応できるサービスを実現することが計画されている。

Linux ディストリビューションが多くのユーザに使用してもらえるようになるためには、タイムリーなメンテナンスサポートを継続的に提供することが必須である。一般的に使用されているディストリビューションは全てパッケージ管理システムを持っており、ユーザはこれによって必要なバグ修正やセキュリティ更新を簡単な手順で適用することができる。このようなメンテナンスサポートがないと、ユーザが全ての OSS アップストリームを監視して適用すべき変更を自分自身で判断した上で適用評価まで行わなければならぬことになり、Linux システム上でのアプリケーション開発にとって非現実的な量の負担となる。我々が手厚いメンテナンスサービスを利用できるのはディストリビュータに所属するその分野のエキスパートの貢献に負うところが大きく、それはある意味属人的なプロセスでもある。IEC61508-3 7.8 Software modification の項目に「ソフトウェアの改変プロセスが適切に定義され透明性を持って運用されること」という旨の内容が語られている通り、機能安全対応のディストリビューション開発においてはアップストリームの監視とバグ修正・セキュリティ更新の適用評価プロセスのトレーサビリティが確保されることが求められる。独自に機能安全対応 Linux ディストリビューションを構築するとなると、数ある OSS プロジェクトを属人的でないプロセスで監視するためには何らかの自動化対策が不可欠となる。

SIL2LinuxMP プロジェクトでは [27](#) ページで述べたように検証プロセスを自動化するための様々なツール・技法が検討されている。図 [3.2](#) は品質管理ライフサイクルの各タスクで適用できる可能性のあるツール・技法の対応付けを示したものである。これを機能安全対応ディストリビューションに拡張するためには、図 [3.2](#) に加えて「アップストリームの監視と適用すべき変更の抽出」をシステムティックに実施する仕組みの導入が必要となる（図 [4.1](#)）。

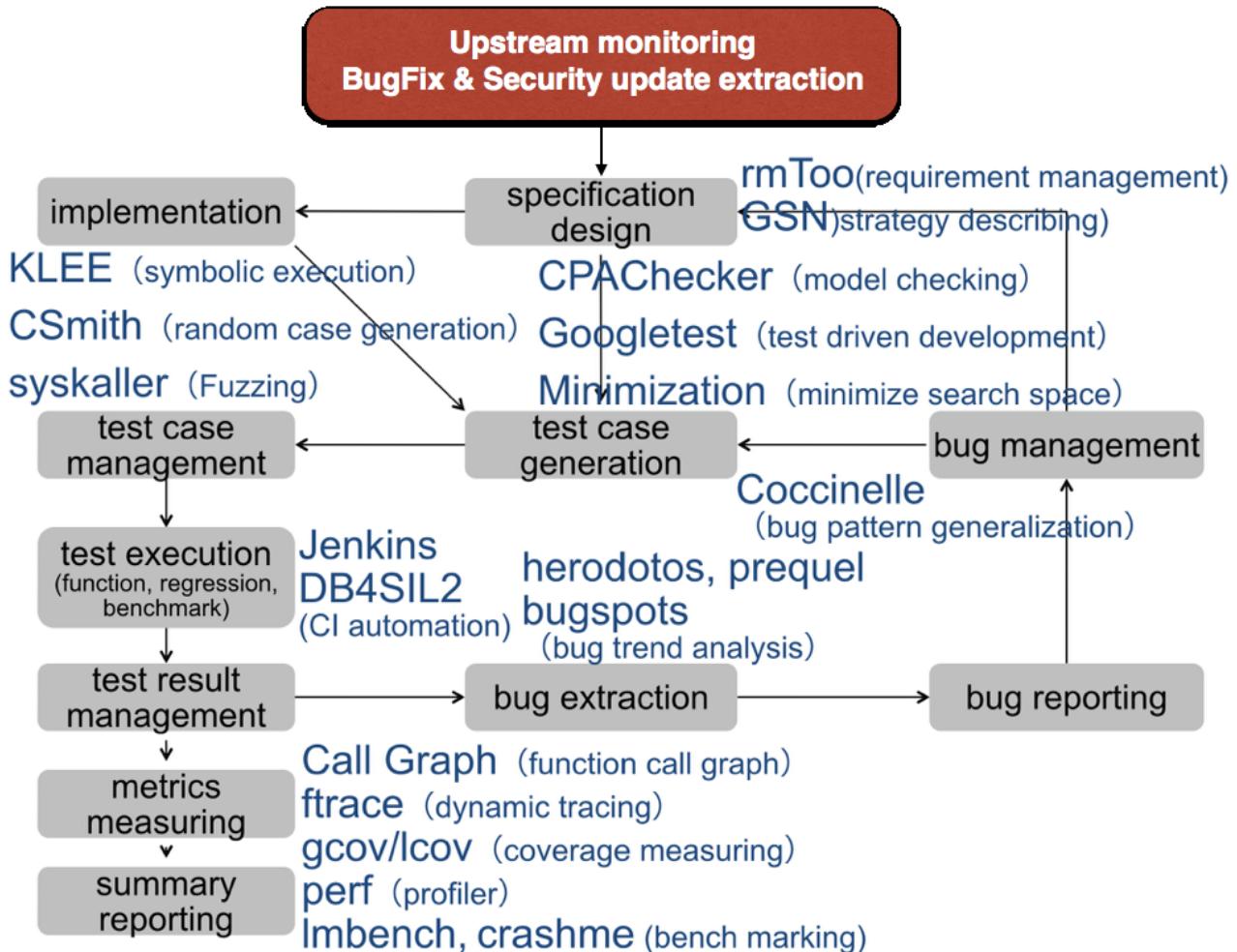


図 4.1 検証フレームワークの機能安全対応 Linux ディストリビューションへの拡張

監視するべき OSS プロジェクトは多数存在するが、それらの開発基盤には事実上の世界標準であるバージョン管理システム Git が共通して使われている。Git には全ての変更履歴がパッチとして記録されており、パッチにはそれぞれ変更の内容・意図・背景に関するコメントが付与されている。アップストリームの監視と適用すべき変更の抽出を自動化するためには、Git パッチに含まれる情報を最大限利用する戦略が考えられる。20 ページでは Git 変更履歴に対してセマンティックパターンサーチを行うツール Prequel を述べた。Prequel の戦略を拡張して、パッチコメントに対するセマンティックパターンサーチを行うことが出来れば例えば特定のレベルのバグ修正やセキュリティ更新を効率よく抽出できる可能性がある。その後の評価や適用判断は人の介在が必要であるものの、一連の活動の契機となるアップストリーム情報のインプットを自動化することで、トレーサビリティの確保された属人的でないプロセス実現の足掛かりを得ることが期待される。

その他、Linux システムのビルドフレームワークとして Yocto Project [32] が広く用いられている。Yocto Project ビルドシステムは Linux ディストリビューションを構成する各 OSS パッケージのメタ情報（バージョン、適用パッチ、依存パッケージなど）から構成されており、各々の OSS パッケージの更新状況は Yocto Project 側のメタ情報にも随時活発に反映されている。OSS アップストリーム情報をイ

ンプットする場所として、Yocto Project のような OSS プロジェクト全体の情報が集まる場を上手に利用できれば、重要なバグ修正やセキュリティ更新を迅速にキャッチしタイムリーに提供する体制を確立できると考えられる。

本活動に対しては、機能安全対応の Linux ディストリビューションの構築と継続的メンテナンスサービス体制を確立することを目標に、図 4.1 に示したような「アップストリームの監視と適用すべき変更の抽出」をシステムマティックに実施する技法および方法論の調査と開発を行う。

4.3 SIL2LinuxMP コミュニティへの継続的なコントリビューション

4 ページで述べた目的の下、これまでと同様に以下の活動を今後も継続して行うことで機能安全に関するノウハウを蓄積し、日立からの貢献を SIL2LinuxMP コミュニティへアピールする。

1. メーリングリスト上で質問や技術提案を行いコミュニティを巻き込んだ議論を行う。
2. ツール開発、実験・調査内容共有、ドキュメントレビューを Git 上で行い貢献の記録を残す。
3. オンライン HAZOP セッションへの参加または開催を行う。

2. のツール開発、実験・調査内容共有は特に DB4SIL2 を対象とする。現在 DB4SIL2 はプロトタイプの段階であり技術調査・検討がさらに必要である。特に、カバレッジ測定技法をユーザ空間にも拡張する具体的方法 (30 ページ)、コンパイラを T3 ツールとして認証するために必要なエビデンスを得る方法論 (10 ページ)、syzkaller (18 ページ) をはじめ fuzzer として利用可能なツールの調査と DB4SIL2 への適用実験が今後の課題として挙げられる。

また、Automotive Grade Linux (AGL) や GENIVI Alliance などの団体が近年の機能安全対応の必要性から SIL2LinuxMP プロジェクトの状況に关心を持っている。今後は外部の OSS コミュニティに対して SIL2LinuxMP の成果とノウハウを共有することが求められるため、開発した検証技術と認証プロセスが現実のアプリケーションに即して適用できる形となることを確実にしていくことが必要である。

第5章

現地の生活や文化の紹介

本章ではインドでの生活や休日の旅行を通して思ったことを中心に記載する。

5.1 平日の様子

筆者が滞在したホテルは HIL オフィスが入居する World Trade Center Bangalore から南に 5km 下った場所にあった。World Trade Center とホテル間の移動にはホテル手配の車を利用した。バスまたはメトロと徒歩で通勤することも可能であったが、HIL の方針で駐在員は研修員含めて仕事関係の移動は全て車を使うこととなっていた。World Trade Center には Amazon 社のオフィスも入居しており、ホテルからの行き帰りの車で Amazon 社員と同乗することがよくあった。話をすると、彼らは新しく Amazon に採用されて一時的にホテルに滞在しているとのことである場合が多かった。複数の Amazon 社員が同乗した場合は、おそらく機密事項に当たるであろうことを人のすぐ隣で大声で情報交換していた。

出勤では当初ホテルを午前 8 時に出発していた。しかしある日突然 World Trade Center へのメインロードが工事を開始したため通勤路を迂回しなければならなくなり、ホテルを出る時間を早めなければならなくなってしまった。その工事は来る日も来る日も一切進んでいない気配がなく、遂に数ヶ月経って筆者が帰国する日になっても地面に大きな穴が空いたままであった。

ホテルや World Trade Center、メトロの駅やショッピングモールの入り口では荷物検査と簡単なボディ検査が実施されている。しかし職員の動きを見ている限りではそれはあくまでも形式上のもので、とても本当の危険物を発見できる体制には思えなかった。建物内にはいたるところに警備員が配置されていた。しかし一見過剰に思える人員配置や諸々の複雑な手続きも、それらの仕事によって雇用が確保されているので、無駄を省き全体の効率を図ろうとする力学がインド社会ではなかなか成立しないようである。

HIL の終業時刻は午後 5 時であり、職場のほとんど全てのインド人が定時で帰宅していた。筆者は日本にいた頃は慢性的に残業をしていたが、HIL は定時で帰るのが当たり前の環境であることと、ホテルの車に予め帰りの時間を指定していることで筆者もほぼ定時で帰るようになった。それに自然と仕事のリズムも合うようになり、定時でキリが良いように仕事が進むようになってそれで業務にも全く支障がなかった。日本に帰っても無駄に残業せずに、基本的に定時で帰る前提で仕事を進めることを継続したい。



図 5.1 World Trade Center とインドの道路。だいたい 100m 間隔で牛に遭遇する。

5.2 週末の様子

日本でやっていたテニスを継続したいと思いバンガロールで練習できる環境を探したところ、インド人ローカルのテニスクラブと日本人会のテニス部を見つけて毎週練習をしていた。平日の移動手段が全て車であり日常生活で運動する機会がホテルのジム以外になかったため、週末のテニスは体力を維持するためにもとても助けになった。ローカルのテニスクラブはコーチとの 1 対 1 レッスンで 1 時間 Rs.600(≈¥1000) と言われ、インド人感覚からするとほぼぼったくり価格であったが、日本でスクールに通うよりはいくらか安いため承知の上で月に一回くらい練習をさせてもらっていた。コーチは毎週毎週「今週末は来るのか?」と電話をかけてくる非常に営業熱心な 19 歳の青年であった。

日本人会テニス部では政府関係の人や企業の駐在員とその家族また学生までが一緒に練習していて、バンガロールに暮らしているたくさんの日本人と知り合うことができた。活動場所はホテルから約 4km 離れた公園 (Cubbon Park) 内にある KSLTA スタジアムであった。ここではプロの試合も行われることがあり、2015 年 10 月には KSLTA で行われた ATP チャレンジーツアー男子ダブルスの決勝を観戦した。練習日時は金曜日 18:30-20:30 と日曜日 10:00-13:00 であった。夜に練習する場合は停電があると照明が復旧するまで 15 分ほど中断してしまうことや、コートが別の予定で突然使えなくなるなど、不便はあったものの日本にいたときよりもはるかにたくさん練習を積むことができていろいろなショットが打てるようになった。そのほかには平日の定時後にコンドミニアム敷地内にあるコートで練習することもあった。



図 5.2 KSLTA スタジアムとインド人テニスクラブ。「今週は行けない」と言っても何度も電話てくる。

日曜日の夕方には人生道場 Toastmasters Club [35] に参加することがあった。Toastmasters International [34] はパブリックスピーチや話し方の上達を目的とする国際的な団体で、バンガロールのコミュニティでは日本語の勉強をしているインド人と英語の勉強をしている日本人と一緒に活動をしていた。活動はバイリンガルで行われていて互いに言語の教え合いをすることがあった。筆者は HIL の駐在員に紹介してもらって参加した。即興で話題を振られて何かを話したり、即興で物語を作ったり、即興で他人のスピーチの論評したりと、とても頭を使う活動であった。



図 5.3 Toastmasters Club バンガロールコミュニティ。筆者よりも日本語の語彙が豊富なインド人がいる。

5.3 交通事情

インドで暮らす上で、日本と全く異なるインドの交通事情に適応することは避けて通れない。タクシーだけではなく様々な乗り物を使えるようになれば行動範囲と自由度がとても広がる。本節では各交通手段別にそれぞれの特徴を述べる。

5.3.1 バス

筆者は休日に移動する際はほとんどバスを使用していた。しかし筆者が聞いた限り、周りの日本人でバンガロールのバスに乗ったことのある人はいなかった。それは、たくさんのバスが街中を走っている中で乗る場所や乗り方また行き先が分からぬからである。HIL のインド人に聞いてもバンガロールのバスは複雑でしかも不安定なので普通使わないと言っていた。

バンガロールには 100 を超える数の路線バスが走っている。それぞれ、V-335EV, 258CC, KBS-3E のような数字とアルファベットの組み合わせが路線名としてバスの先頭に表示されている。路線名の表示は手書きだったり、電光であっても消えていたり、現地語で書かれたりするため、まず路線名の視認が困難

なことが多い。路線名は走る地域や行き先によって似たような数字が割り当てられているようであるが、その規則性は全く不明である。乗るべき路線名は Google Map で行き先を調べることで分かるものの時間は全く当てにできない。Google Map では通常複数の路線名が検索結果に出てくるので、それらを全てメモしておくとバスに乗れる確率が高まる。なぜなら、Google Map で示されたバス停に行っても狙った路線名のバスが一定時間内に来るとは限らないからである。バス停といつても時刻表やバス停名などといった親切なものは一切なく、かろうじて屋根と椅子があることでバス停と分かるものとなっている。目印が全くなく人が集まっているだけのバス停もある。特に僻地のバス停ほどバスが来る頻度が低いため、乗ることのできる路線名をできる限り多く把握しておくことが必要となる。筆者の場合はホテルから徒歩 15 分程度の場所にバンガロールシティ駅に隣接するバスターミナル (Kempegowda Bus Station) があったので、バスの乗り降りはそのバスターミナルで行った。ここでは近辺の全ての路線が乗り入れるので、乗りたい路線にはほぼ確実に一定時間内に乗ることができた。

バスの料金は、乗ってからしばらくしてやってくる集金係に行き先を告げると金額を教えてくれるので手渡しで支払う。親切な集金係だと伝えた場所に着くと知らせてくれる場合もあるが、基本的に自分で降りる場所を把握していないなければならない。気の利いた車内アナウンスなどはもちろんない。料金は 5km-15km の道のりでだいたい Rs.10-20(≈ ¥15-30) と非常に安い。ここで Rs.50 紙幣や Rs.100 紙幣を出すと受け取ってもらえないか非常に嫌がられるので、Rs.1 や Rs.10 単位の細かい額を持っておかないと支払いに手間取ってしまう。Rs.500 紙幣や Rs.1000 紙幣などは言語道断である。どうしても集金係にお釣りがないときは、「降りるときに何ルピー受け取る」というメモをその場で書いて渡してくれる。もちろん申告しなかったり降りるときもお釣りがなかったりという場合はもらうことができない。バスに限らず商店やモールでもみんなが細かいお金を欲しがるので、気をつけて小額紙幣や小銭を切らさないようにしていないと Rs.500 札や Rs.1000 札だけになってしまい日常生活に支障をきたすことになる。

バスには必ず運転手と集金係の必ず二人のスタッフがいる。バスがどんなにぎゅうぎゅう詰めに混んでいても、集金係は新しく乗ってきた人を見失わずに人を押し退けバスの中を前に後ろに移動するのでとても大変である。バスの中は前半分が女性で後半分が男性の空間という暗黙の了解があり、空いているからといって前に座ると注意される。バスは走行中もドアが常に開きっぱなしで、ドアが壊れていることもよくある。バス車体の状態は整備という概念があるのか疑問に思うほどで、どのバスもひどい排気ガスを放出していた。道路が平らでないことも相まって乗り心地は良いとは言えないが、タイミングよく乗ることができれば非常に安価で便利な移動手段である。ただし道路が渋滞しているときは徒歩よりも遅い。



図 5.4 バンガロール市内の路線バス内と Kempegowda Bus Station 内の案内一つ

5.3.2 鉄道

インドの鉄道には主に都市間や長距離を結ぶディーゼル機関車と、デリーなどの都市圏内で運行するメトロと言われる電車がある。ここでは中長距離を結ぶ Indian Railways について記載する。筆者は週末や休日の旅行の際に Indian Railways を利用した。インド滞在中に利用した区間は Mumbai-Aurangabad 間（約 360km、8 時間）、Bangalore-Mysore 間（約 140km、3 時間）、Bangalore-Hospet 間（約 350km、8 時間）であった。Indian Railways を利用するためには座席の予約が必要であることになっている。ただし後述するように事実上は予約せず運賃も払わずに乗車することは可能であるため、無賃乗車で利用している客は一定数存在すると思われる。筆者は MakeMyTrip というインドの旅行手配サービス [23] を日本人会テニス部で教えてもらい、そこを介して Indian Railways チケットの購入を行った。駅の窓口で直接チケットを購入することもできるようであるが、Indian Railways の座席予約システムが非常に複雑であることもありインド人と直接会話してチケットを確実に確保できる自信がなかったため web 上で予約を行った。

Indian Railways チケットを購入するためには、前提として Waiting List という Indian Railways 独自のシステムを理解する必要がある。Indian Railways にはこの複雑な Waiting List システムに起因する数々の独自のしきたりが存在しており、特に旅行者が Indian Railways を利用する際に混乱する要因となっている。web 上を検索するとこの Indian Railways 予約方法に関する Waiting List 用語の質問や解説が山のように出てくるほか、[Indian Railways の利用方法ノウハウを第三者がまとめたサイト](#) [31] では Waiting List の仕組み解説のために特別にページを割いているほどである。大前提として、Indian Railways チケットは購入した時点で座席が確保されない。座席数を超える予約が入っている場合であっても Indian Railways 予約システム上はチケットを購入できてしまうため、それをもって実際に座席を確保できる条件を予め把握しておかないと、予約が確定したと勘違いした結果当日になって列車に乗ることが出来ないということが起こり得る。

用意された座席数が埋まっていない場合は座席の権利が確定したチケット (confirmed) を購入できるが、既に埋まっている場合は Reservation Against Cancellation (RAC) または Waiting List と呼ばれる区分のチケットを購入することになる。RAC は「列車に乗車する権利があるが座席は保証しない」チケットである。この場合は座席の権利を持った客が現れない場合そこに座るか、他の客の席を分けてもらうか、または立って乗車することになる。confirmed チケットのキャンセルがあると隨時先着順で RAC が confirmed に繰り上がる。confirmed と RAC で割当てられた分のチケットが売り切れると Waiting List チケットを購入することとなる。Waiting List は座席の権利も乗車の権利もなく、confirmed チケットまたは RAC チケットでキャンセルがあることを前提に権利が繰り上がることが期待できるのみである。当日になっても Waiting List から繰り上がりなかつた場合は料金が自動的に返金される。RAC または Waiting List は購入する時点で何人たまっているのかが分かるようになっており、購入した後もキャンセル状況によって自分がリストの何番目なのかの情報が MakeMyTrip サイト上で隨時更新されるため、乗車・座席の権利が繰り上がる可能性をある程度予測することができる。Waiting List チケットはさらに 4 種類ほどのタイプに区分されるがここではこれ以上の説明を省略する。どの区分のチケットであっても期日前なら特にペナルティなくキャンセルが可能である。ただし MakeMyTrip や Indian

Railways のサイトは特に夜間不安定になる傾向があるため、予約手続きが途中で失敗したりサイトが一時閉鎖していたりということに頻繁に遭遇する。Waiting List の仕組みを理解し、かつ MakeMyTrip 上の手続きが成功して初めて何らかの乗車可能性が期待できるチケットを購入することができる。

当日になって confirmed チケットを手にしていた場合であっても座席の場所は列車の発車時刻数時間前に用意される Chart を確認するまで把握することが出来ない。Chart はその列車に乗車できる客の最終確定リストであって、乗車直前に MakeMyTrip で確認するか、駅の端末または張り出された紙で確認するか、各車両のドアに張り出された紙を確認する必要がある。Chart には客の名前や性別年齢等の個人情報がプライバシー無関係に記載されている。当日になって Chart に自分の名前があることと座席番号が記載されていることを確認して初めて安心して列車に乗ることができる（図 5.5）。

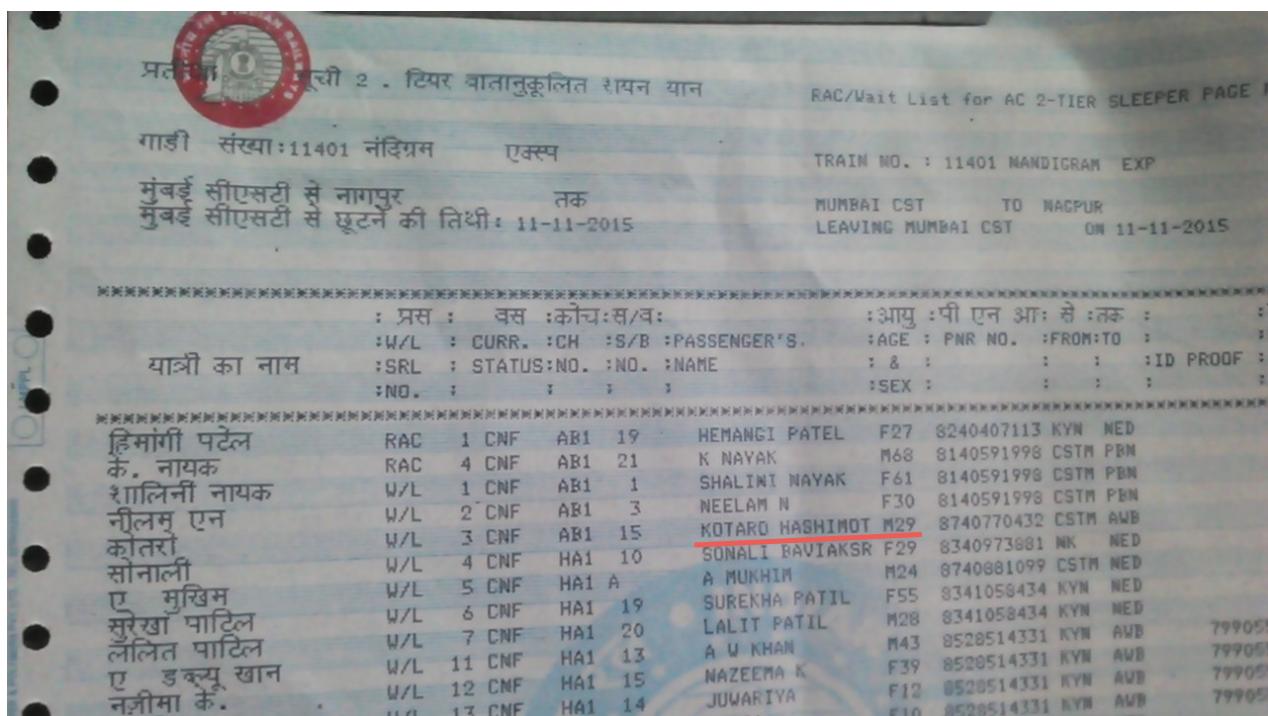


図 5.5 列車の入り口に貼られている Chart に自分の名前が確認できて安心しているところ

このようなキャンセル待ち前提のシステムは旅行の計画を立てる上で大きな不安要素となるため、筆者は confirmed で取れるチケットを探して予約を行い、当日も無事に Chart に名前を確認して座席を確保することができた。しかし実際に乗車する現場では、特に下位クラスの車両では予約したはずの座席に既に人がいたり、網棚に人が登るほど乗車率が高かったり、特にチケットを確認されなかつたりと、複雑な予約システムに苦労して従う意味が本当にあったのか疑問である。

ちなみにキャンセル前提でとりあえず予約するという文化は HIL 内にも存在しており、HIL のミーティングルームの予約はほぼ全て常に誰かに押さえられていた。実際は予約されたミーティングルームは使われていないことが多いため、予約システムの状況関係なく隨時空いている部屋を探して利用することがほとんどであった。



図 5.6 Chart や予約システムなど関係なく大量に乗車するインド人

5.3.3 メトロ

メトロは都市圏内で運行される電車である。Indian Railways の中長距離列車の場合では、だいたい常に 30 分くらい遅れて、走行中ドアが開きっぱなしで、停車の案内などが一切なく改札もない、ということと比較するとメトロはきちんと管理されている。利用する場合は、駅の窓口で行き先を告げてトークンと呼ばれる非接触 IC タグを受け取る。料金はだいたい 5km で Rs.15(≈ ¥25) である。プリペイドカードを持っていれば毎回窓口に行く必要がない。メトロは分単位のスケジュールで運行されており本数も多く、インドでは時間が信頼できる唯一の交通手段である。日本の電車とほぼ相違ない。

ただしバンガロールでは現在もメトロは建設中であって、部分的に完成している範囲しか運行していない。数年単位の計画の遅れはインドでは特に気にしないようである。全て完成するとバンガロール市内を中心に東西と南北に交差するようになり非常に便利になるが、筆者の滞在中は交差するところまではるかに届いていなかった。メトロ一本で目的地に行けることはないため徒歩と組み合わせて利用していたが、朝夕の道路渋滞時にバスが当たにならないときなどは便利であった。



図 5.7 ショッピングモールが隣接するメトロ駅。ここから先の線路は工事中である。

5.3.4 リキシャ

リキシャはインド全国で見かける個人タクシー相当の三輪の乗り物で、「オート」や「トゥクトゥク」と言うこともある。バンガロールでは四輪車のタクシーが走っていないため、現地の人も旅行者もリキシャを使う機会がとても多い。料金はだいたい路線バスの3-4倍程度である。ただし、ドライバーによってはメーターを使わずに高い料金をふっかけてくることがある。そういう場合は無理して交渉せずさっさと降りて別のリキシャに声をかければよい。交渉を避けてスムーズにメーターを使ってもらうためには、現地の地理をよく知ってるフリをして目的地を口頭で伝えることが重要である。ここで地図を出して行き先を伝えようとしてもリキシャのドライバーには地図が通用しない。地図ではなく道路名や目印になる建物や場所を言うとスムーズである。むしろ地図を出すと道や相場を知らない観光客と思われてカモにされる。後々の面倒を避けようと乗るときに「ここまでいくらか」と聞くのも、メーターを知らない観光客と思われて自分からぼったくられに行くようなものなので、メーターを使ってくれるように自然に振舞うことが重要である。あっさりメーターを使ってくれない場合は遠慮せず降りて他のリキシャを探せばよい。



図5.8 インド全国で気軽に乗れるリキシャ。上半分が黄色で下が緑か黒のデザイン。

5.3.5 タクシー

インドでは流しのタクシーが走っていない。タクシーを使う場面はホテル手配の車で通勤するときと職場手配の車でどこかに出かける場合のみであった。タクシーを使う場合はドライバーに時間と場所を毎回指定する必要がある。職場から帰るときもWorld Trade Centerに来て欲しい時間を毎回伝えていたが、時間通りに来てくれる確率は非常に低かった。インドでは日本と異なり時間が正確に守られない事象に頻繁に遭遇する。一部のインド人はそもそも人を待たせることを悪いと思っていないように見受けられるので、それでいちいち怒らずにうまくイナす心がけが重要である。しかし、毎日事前に時間を伝えて来てもらうことと、伝えた時間に来ないことを考えると、移動のためにそのような不確実で手続きが面倒なインフラに頼らざるを得ないことが非常に不自由に感じるようになった。

5.3.6 徒歩

以上の交通事情を総合すると、渋滞や時間・人の不確実さに依存しない移動手段として、徒歩が最も楽で確実で自由度が高いと思うようになった。そのため、5km程度の移動では特にバスを調べずに歩くようになってしまった。再度インドに長期滞在することがあったらこんどは移動を自己完結できるように自転車を用意したい。ただし自転車が盗まれないための何らかの対策は必要である。また、バスやリキシャが出す排気ガス対策としてマスクも必須である。インドでは自転車は貧困層の乗り物という意識があると聞いたことがあるが、バンガロールでは自転車に乗る人はほとんど見かけなかった。

バンガロールの道路は一日中常にクラクションの音であふれている。HIL通勤時の車でのドライバーの動きを見ている限り、クラクションは危ないときに鳴らすのではなく「通りますよ」の意味で自分の存在を知らせるために鳴らすらしい。日本ではクラクションの意味が重いため鳴らすのに一定の心理的ハンドルがあるが、インドでは追い越しするときや細い道に入るときや人のそばを通るときなど非常に気軽にクラクションが鳴らされる。大きなトラックの後ろにはわざわざ”Sound Horn”と書かれており、鳴らされる方もクラクションを期待しているようである。おそらくインドの道を自転車で走るとなると追い越されるたびにクラクションを鳴らされてびっくりすることになるが、それは別に危ないと怒られているわけではないので、クラクションをいちいち気にしない強いメンタルを身につける必要がある。

5.4 旅行

研修期間中は休日や週末を利用して南インド中心に旅行に出かけた。年末年始はスリランカの街と遺跡を巡り、3月にはドイツに出張しハイデルベルグの街を観光した。

5.4.1 ムンバイ、オーランガーバード、エローラ・アジャンタ

11月中頃に Diwali というインドの祭典があり、それに伴い5日間の連休を取得して世界遺産に登録されているエローラ石窟群とアジャンター石窟群を訪れた。バンガロールから飛行機でムンバイまで行き、ムンバイ市内をローカル鉄道で移動し、ムンバイからオーランガーバードまで7時間の長距離列車に乗り、オーランガーバードからエローラとアジャンターまでは長距離バスを使った。旅行の目的の半分以上が、インド国内の特に初めての鉄道での移動に挑戦することであった。先述した通りインドの鉄道は乗り方が複雑なため、移動中に迷ったり乗り換えを間違えたりしても大丈夫なようにかなり余裕をもったスケジュールを組んだが、しかし実際は拍子抜けするくらい予定通りに移動ができてしまった。ムンバイの空港から鉄道のターミナル駅に行く途中では乗り合わせた現地人が付き添って乗り換えの仕方まで教えてくれた。特にそれでチップを求められるわけでもなかった。オーランガーバードへ向かう長距離列車では互いに口の訊けない子連れの夫婦と相席になり、長距離バスでは高校生達と連絡先を交換し、アジャンターで一泊したときはたまたま知り合った現地の人に家と街を案内してもらい、そのほか行く先々でたくさんの現地の人との出会いがあってその度に何かと助けてもらった。

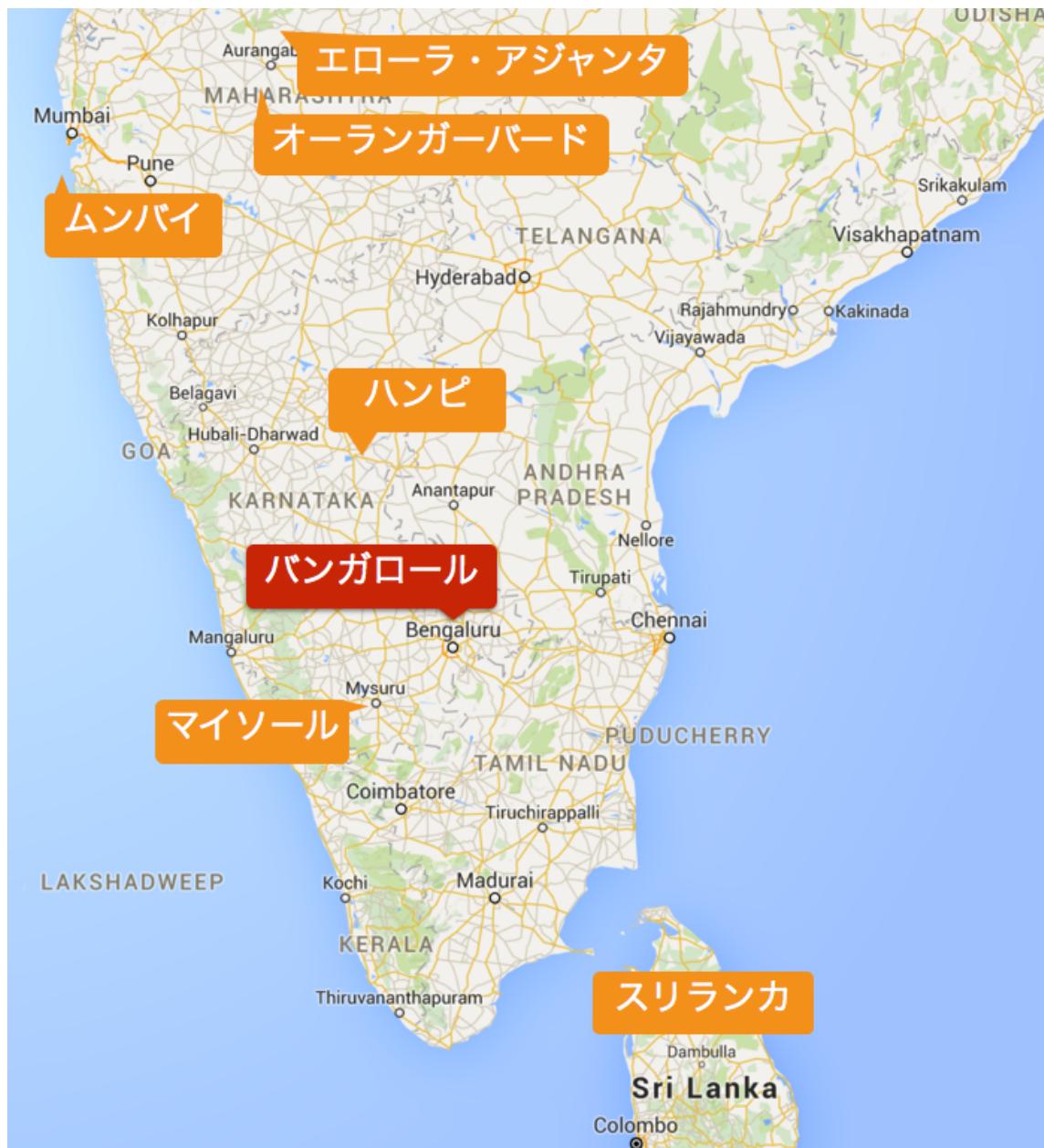


図 5.9 研修期間中に訪れた南インド周辺の地域



図 5.10 ムンバイの世界遺産であり現役ターミナル駅の Chhatrapati Shivaji Terminus

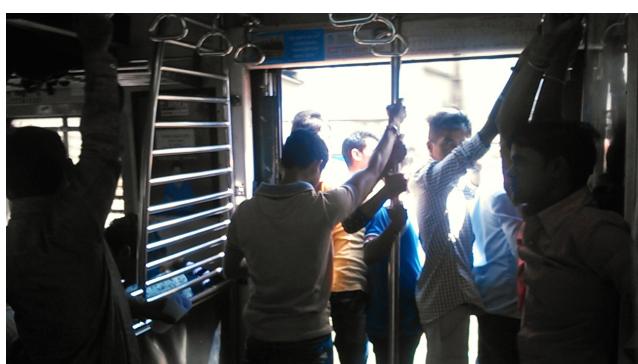


図 5.11 ムンバイ市内の鉄道。走行中も開きっぱなしのドアから身を乗り出している人が多い。



図 5.12 オーランガーバードのバスタークナール。
世界的な遺跡への拠点なのに英語の案内がない。



図 5.13 タージマハルによく似た Bibi Ka Maqbara。行く先々で現地人に写真をせがまれる。



図 5.14 Diwali 期間中に民家の玄関口
に描かれる装飾



図 5.15 アジャンター石窟群に隣接す
る街 Fardapur を案内してくれた青年



図 5.16 Fardapur
青年の家と家族

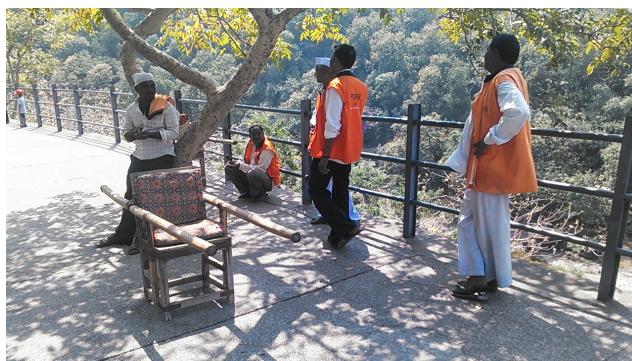


図 5.17 アジャンター石窟群での「椅子」客待ち



図 5.18 駅に止まるたびに売り子がやってくる

5.4.2 マイソール

マイソールはかつてマイソール王国の首都として栄えた文化的歴史がある都市である。バンガロールから日帰りで片道約3時間かけて鉄道で移動し、マイソール宮殿やマーケットを見て歩いた。マーケットで

は大量の生花、野菜、果物（ほとんどバナナ）、粉、布、香辛料、茶、葉っぱ、草、人の髪などが取引されており、何のためにどうやって使うのかが分からないものが多かった。



図 5.19 マイソールの Devaraja Market



図 5.20 Diwali や Holi の祭典で使われる粉

5.4.3 ハンピ

ハンピはカルナータカ州北部にある都市遺跡である。あまり観光地されておらず、中心部であってもごく小さなバザールや安宿街があるだけのとても静かな場所であった。バンガロールからは約8時間の寝台列車でホスペットまで移動し、そこから約1時間のバスで行くことができる。観光客にはインド人のほか欧米人が多かった。訪れた時期が3月下旬でちょうどインドの祭典Holiの時期だったため、顔や体にカラフルな粉や液体をかけた（かけられた）人をたくさん見た。バザールを歩いていて突然筆者のすぐ後ろで現地の子供に色水をかけられた観光客がいたが、筆者は防水対策をしていなかったためすぐに逃げた。ハンピには広い範囲に遺跡が点在していて、地域全体に奇妙な形の岩が奇妙な構成で積み上げられている。人工物なのか自然現象の結果なのか分からないものが多く、いったい何がどうなればあのような岩の構成物ができるのか全く理解できなかった。

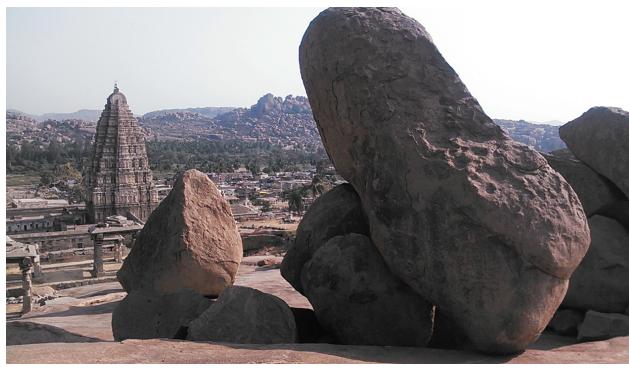


図 5.21 ハンピで唯一遺跡にならずに現役で活動し続ける寺院 Virupaksha Temple

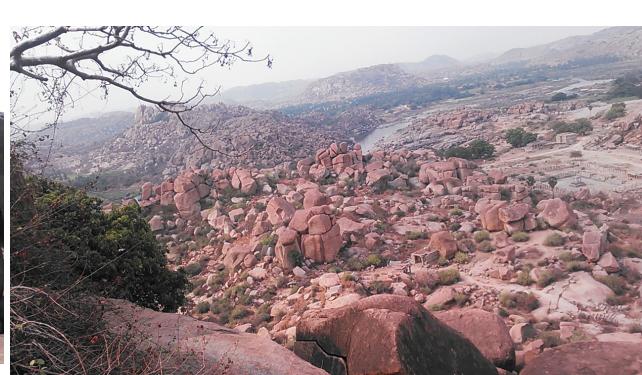


図 5.22 Matanga Hill からの風景。ここで昼寝した。



図 5.23 遺跡でピクニックをするインド人



図 5.24 喜んで写真に応じてくれるが一回 Rs.100



図 5.25 Holi を祝う Hampi の子供



図 5.26 Holi された犬

5.4.4 スリランカ

スリランカはインドのすぐ南に位置しており、バンガロールから気軽に気軽に行くことができる。スリランカ旅行を計画した当初はアーユルヴェーダ施設に一週間入ることを考えていたが、スリランカは初めてだったため王道の遺跡・寺院巡りをすることとした。バンガロールからネゴンボまでは飛行機で行き、そこから各都市への移動には鉄道と長距離バスを利用した。スリランカの鉄道システムは Indian Railways のように複雑でないので、特に予約せずとも問題なく現地でチケットを購入できる。ちなみにインドの隣国であるにもかかわらずスリランカの空港 (Bandaranaike) ではインドルピーの両替ができない。これはインドルピーが国外持出し禁止とされているからであるが、タイ、シンガポール、マレーシアなどでインド人の移民が多く集まる地域では両替が可能な場所が存在するらしい。スリランカでも探せばコロンボでインドルピーを受け付けてくれる両替商があるかもしれない。

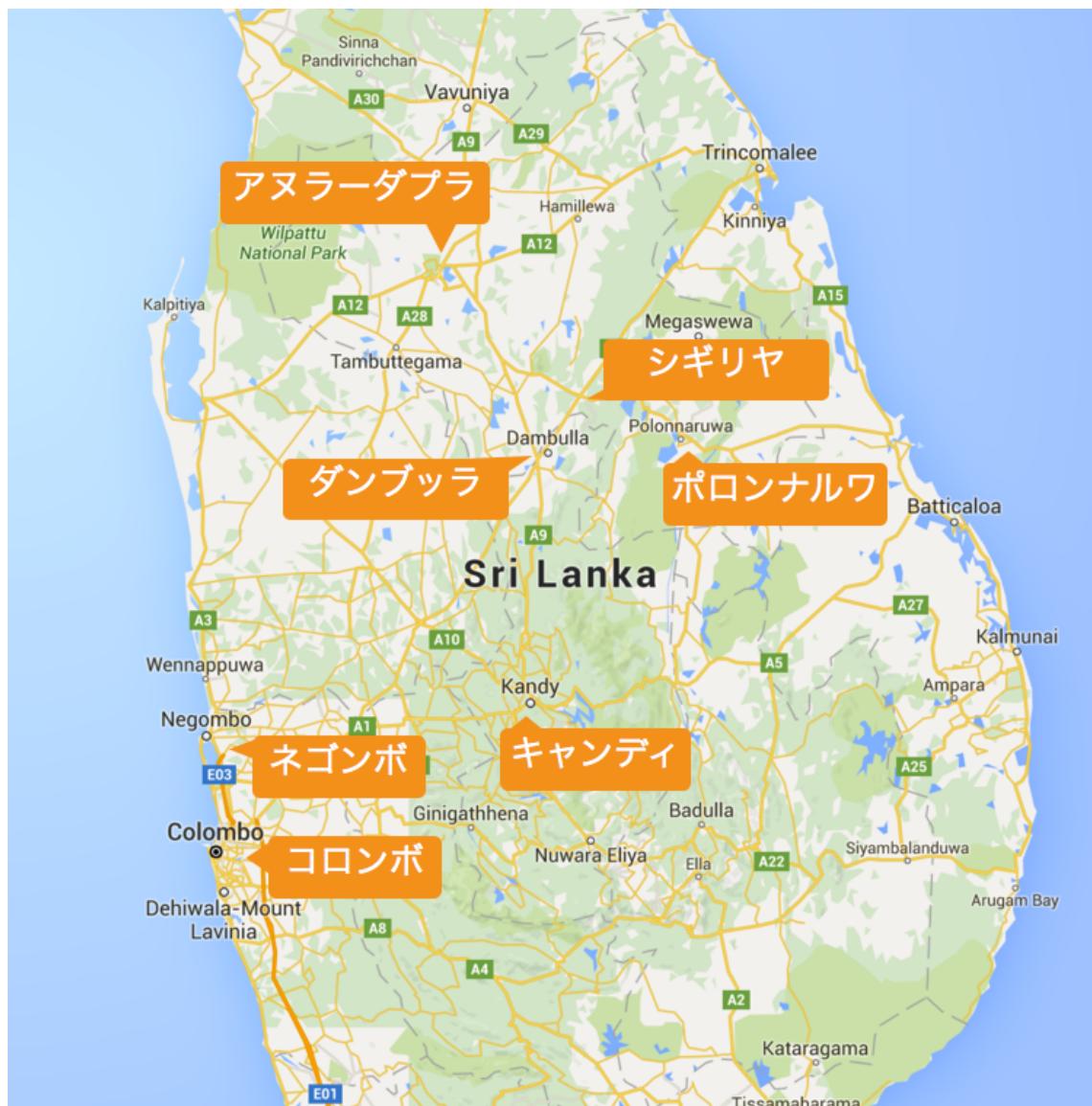


図 5.27 年末年始に訪れたスリランカの地域



図 5.28 コロンボから各都市への起点となる Fort 駅

図 5.29 アヌラーダプラに到着したところ



図 5.30 都市間を結ぶバス。本数が多くて便利。

図 5.31 売りに出されているトウクトゥク。
Alpine 社との関係は不明。

図 5.32 どこにでも必ず置いてある AJINOMOTO



図 5.33 ポロンナルワ遺跡群に住み着いている猿



図 5.34 キャンディ仏歯寺とその隣の池



スリランカはインドと比べると道が綺麗で、また旅行者に対して優しいという印象を持った。現地に行っても必要な情報がきちんと英語で表示されていて、迷っていたら誰かがすぐに声をかけてくれて、かといって客引きがしつこいわけでもなく、旅の難易度はインドよりも低いと感じた。インドとは地理的に近く文化も似ている点が多いが、国教が仏教であるため遺跡や寺院では仏像を多く見る。



図 5.35 ゲストハウスで借りた自転車が日本の寄付物のようで「阿倍野郵便局」と書いてあった



図 5.36 高さ 200m のシギリヤロック



図 5.37 寺院の隣で売られているハスの花

5.4.5 ドイツ

3月第一週に SIL2LinuxMP マイルストンミーティングがドイツ・ハイデルベルグで行われた。バンガロールからはフランクフルトまで直行便を使い、フランクフルトからハイデルベルグまでシャトルバスで移動した。この時期のドイツはバンガロールと 30 °C以上の気温差があったが、インドには半袖と薄い服しか持てこなかつたためそのままの格好で行ったところフランクフルト空港は雪が降っていてとても寒かった。SIL2LinuxMP ミーティング終了後に参加者数人でハイデルベルグ城と Old Town の観光を行った際、同行した人があまりに寒そうだと服を貸してくれた。ハイデルベルグの古い街並みはとても美しく機能的だという印象を持った。鉄道とトラム網がとても発達していて、道路は自転車が快適に走れるよう整備されていた。



図 5.38 ハイデルベルグ街中を走るトラム



図 5.39 SIL2LinuxMP ミーティング会場



図 5.40 ハイデルベルグ城とそこから見下ろす Old Town



5.5 食事

インドではほとんど毎日果物を食べて生きていた。あまり外の屋台で食べ物を買うなどのリスクを冒さなかったので、日本に帰るまで体調を崩すことはなかった。筆者は日本では朝1リットルくらいブラックコーヒーを飲む。しかしインドではブラックコーヒーを飲む習慣がなく、ホテルでもコーヒーはミルク入りで用意されている。そのため毎朝ブラックコーヒーをオーダーしていた。外国人も泊まるホテルで他にもブラックコーヒーをオーダーする人は多くいたにもかかわらず、わざわざオーダーしないと用意してくれないシステムが改善される気配は遂になかった。インド人は大半がベジタリアンであるため肉や魚を使ったメニューが少ない。提供される肉も一般的な店やホテルではチキンのみである。牛肉や豚肉を提供する店もバンガロール市内にはないことはないがごく限られている。ホテルでは毎日野菜がたっぷり朝食に並んでおり、半年間野菜と果物を主食としていたため日本にいたときよりも食生活は健康的であった。日本でいうカレーに相当するものはあまり口にしなかった。インド料理のことはよく分からぬが、インドはカレーと一括りで言うことが失礼なほど多様なカレーのようなものがある。むしろだいたい何を食べても固形物でも液体でもカレーのような味のするスパイスが効いているため、おそらくインドはカレーという区別がないのではないかと思う。もちろん日本で食べるカレーの味をインドで求めるならば日本食レストランに行かなければならない。行ってもあるかどうかは分からない。研修期間の後半になるにつれて、日本に帰ったら白米と海鮮物をとても食べたいと思うようになった。

5.6 英語

インド人の英語は聞き取りづらいというステレオタイプがあるが、発音は人それぞれである。初めは聞き取りづらい人であっても耳が慣れれば問題なく意思疎通できるようになる。HILで働いていたインド人は英語のネイティブと遜色ないほど聞き取りやすく流暢な発音をする人が多かった。筆者と一緒に仕事をしていた Krishnaji さん、Geet さんはそれぞれアメリカ、イギリスでの長期滞在経験があるとのことであった。オフィスの外ではリキシャの運転手や特に地方に旅行に出かけたときに英語が通じにくいくことがあった。筆者は口頭でも文面でも結論や要点を先に伝えることを意識し、特に文法が正しくなくてもキーワードとなる単語を先に言うことを重視していた。

バンガロールで耳にするまたは目にすることの多かった特徴的な英語表現としては、「営業時間」や「都合の良い時間帯」を意味するときに”timing”と言うことや、文中で重複を避ける際の代名詞に”the same”を使うことや、特にインド人同士の会話で”~, right ?”のようなニュアンスで”~, ノウ ?”と言うものがあった。

謝辞

海外業務研修を実施するにあたり非常に多くの人に手厚いサポートをしていただきました。派遣を推薦してくださった上長と研修を後押ししていただいた職場の皆様、渡航前の準備から帰国後まで全面的に手続きを手伝っていただいた総務、派遣を受け入れていただき現地の生活をサポートしてくださった HIL の皆様、共に SIL2LinuxMP プロジェクトに取り組んでいる研究所と HIL のメンバ、OSADL, OpenTech の技術者、毎日職場まで運転してくれたドライバーと日々の生活の面倒を見てくれたホテルのスタッフ、楽しい休日の思い出を作ってくれた日本人会テニス部と Toastmasters Club、旅先でフレンドリーに接してくれていろいろ助けてもらった見ず知らずの現地の人々、非常に多くの人のサポートがあってこの業務研修ができたことを感謝しています。今後も継続して SIL2LinuxMP プロジェクトに取り組み、オープンソースソフトウェアの機能安全対応プロセス研究と品質保証ノウハウ蓄積を進めます。特に成果を HiICS のビジネスに展開することを念頭に置いて、引き続き SIL2LinuxMP コミュニティと協力していく所存です。

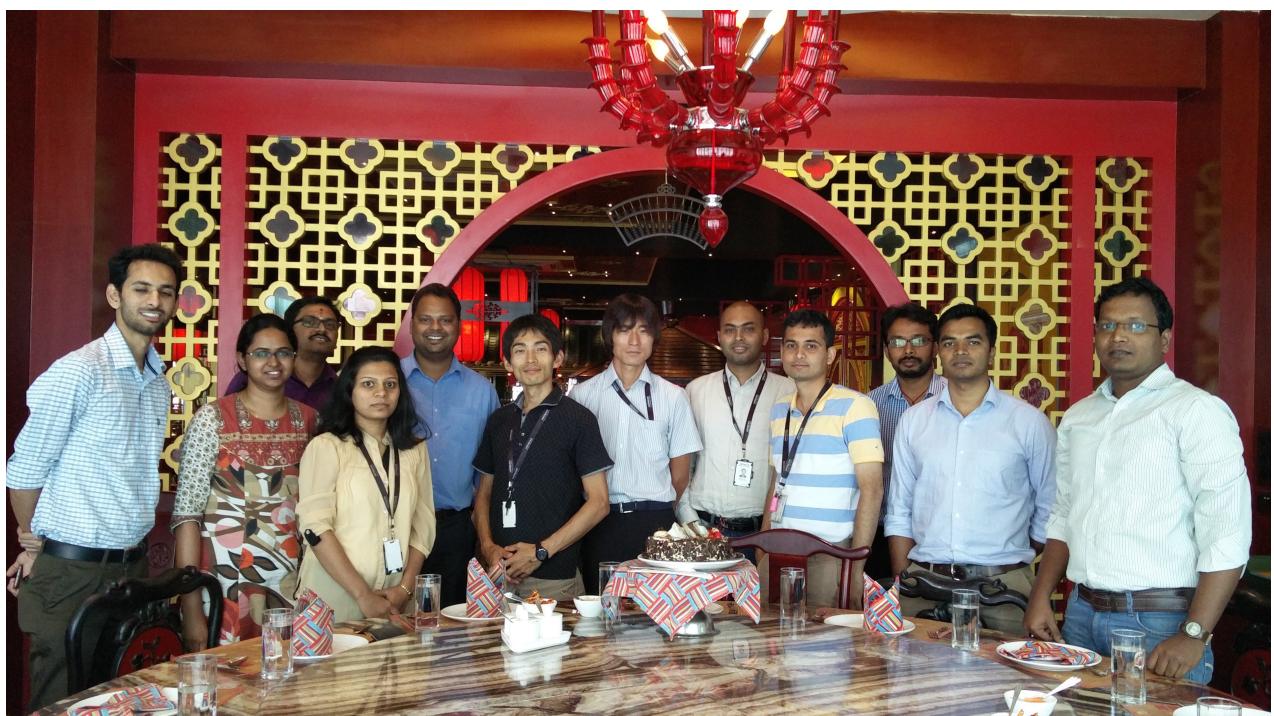


図 5.41 HIL で一緒に仕事をしたチーム

参考文献

- [1] -fdirectives-only question (nicholas の gcc コミュニティへの質問) . <https://gcc.gnu.org/ml/gcc-help/2015-05/msg00012.html>.
- [2] Advocate. <http://ti.arc.nasa.gov/m/profile/edenney/papers/sassur2012.pdf>.
- [3] Assurance case editor(acedit). <https://code.google.com/p/acedit/>.
- [4] Automated program verification environmen(aprove). <http://aprove.informatik.rwth-aachen.de>.
- [5] bugspot. <https://github.com/igorigorik/bugspots>.
- [6] Coccigrep. <https://github.com/regit/coccigrep>.
- [7] Coccinelle. <http://coccinelle.lip6.fr/>.
- [8] Coccinellery: A gallery of semantic patches. <http://coccinellery.org/>.
- [9] Code minimization の実装. <https://github.com/Hitachi-India-Pvt-Ltd-RD/minimization>.
- [10] Codeviz. <http://www.csn.ul.ie/~mel/projects/codeviz/>.
- [11] Competition on software verification(sv-comp) の総合結果. <http://sv-comp.sosy-lab.org/2015/results/index.php>.
- [12] Complexity: Measure complexity of c source. <https://www.gnu.org/software/complexity/manual/complexity.html>.
- [13] Cpachecker. <http://cpachecker.sosy-lab.org/>.
- [14] Csmith. <https://embed.cs.utah.edu/csmith/>.
- [15] Dcase. <http://wiki.portal.chalmers.se/agda/pmwiki.php?n=D-Case-Agda.D-Case-Agda>.
- [16] git history statistics generator(gitstats). <http://gitstats.sourceforge.net/>.
- [17] Github に主戦場を移した世界の it 人材獲得競争 the software, be open or die. <https://wirelesswire.jp/2016/01/49267/>.
- [18] Google が 2011 年に公開したバグ予測アルゴリズム. <http://google-engtools.blogspot.sg/2011/12/bug-prediction-at-google.html>.
- [19] Gsn community standard version 1. http://www.goalstructuringnotation.info/documents/GSN_Standard.pdf.
- [20] Gsn(goal structuring notation) 解説. <http://blogs.itmedia.co.jp/hiranabe/2013/11/goal-structuring-network.html>.
- [21] Herodotos. <http://coccinelle.lip6.fr/herodotos/docs/herodotos.html>.
- [22] Klee. <https://klee.github.io/>.

- [23] Makemytrip. <http://www.makemytrip.com>.
- [24] ncc. <http://students.ceid.upatras.gr/~sxanth/ncc/>.
- [25] Preprocessor output - the c preprocessor. <https://gcc.gnu.org/onlinedocs/cpp/Preprocessor-Output.html>.
- [26] Requirement management tool(rmtoo). <http://rmtoo.florath.net>.
- [27] Semantic patch の文法. http://coccinelle.lip6.fr/docs/main_grammar.pdf.
- [28] Sil2linuxmp メーリングリスト. <http://lists.osadl.org/pipermail/sil2linuxmp/>.
- [29] Strip linux kernel sources according to .config (stack overflow の記事). <http://stackoverflow.com/questions/7353640/strip-linux-kernel-sources-according-to-config>.
- [30] syzkaller. <https://github.com/google/syzkaller>.
- [31] Train stuff in india. <http://trainstuff.in/>.
- [32] Yocto project. <https://git.yoctoproject.org/cgit/cgit.cgi/>.
- [33] サイボウズの企業風土. http://www.slideshare.net/chika_nakazawa/ss-27726686.
- [34] トーストマスターズ・インターナショナル. <https://ja.wikipedia.org/wiki/トーストマスターズ・インターナショナル>.
- [35] 人生道場 toastmasters club. <https://www.facebook.com/jinseidojo/timeline>.