ソフトウェアテストシンポジウム 2009

実証！
コベリティのソースコード静的解析ツールと構造分析ツールがもたらすソフトウェア開発効率の最適化

コベリティ 日本支社

リージョナル セールスマネージャー

丸山　智樹

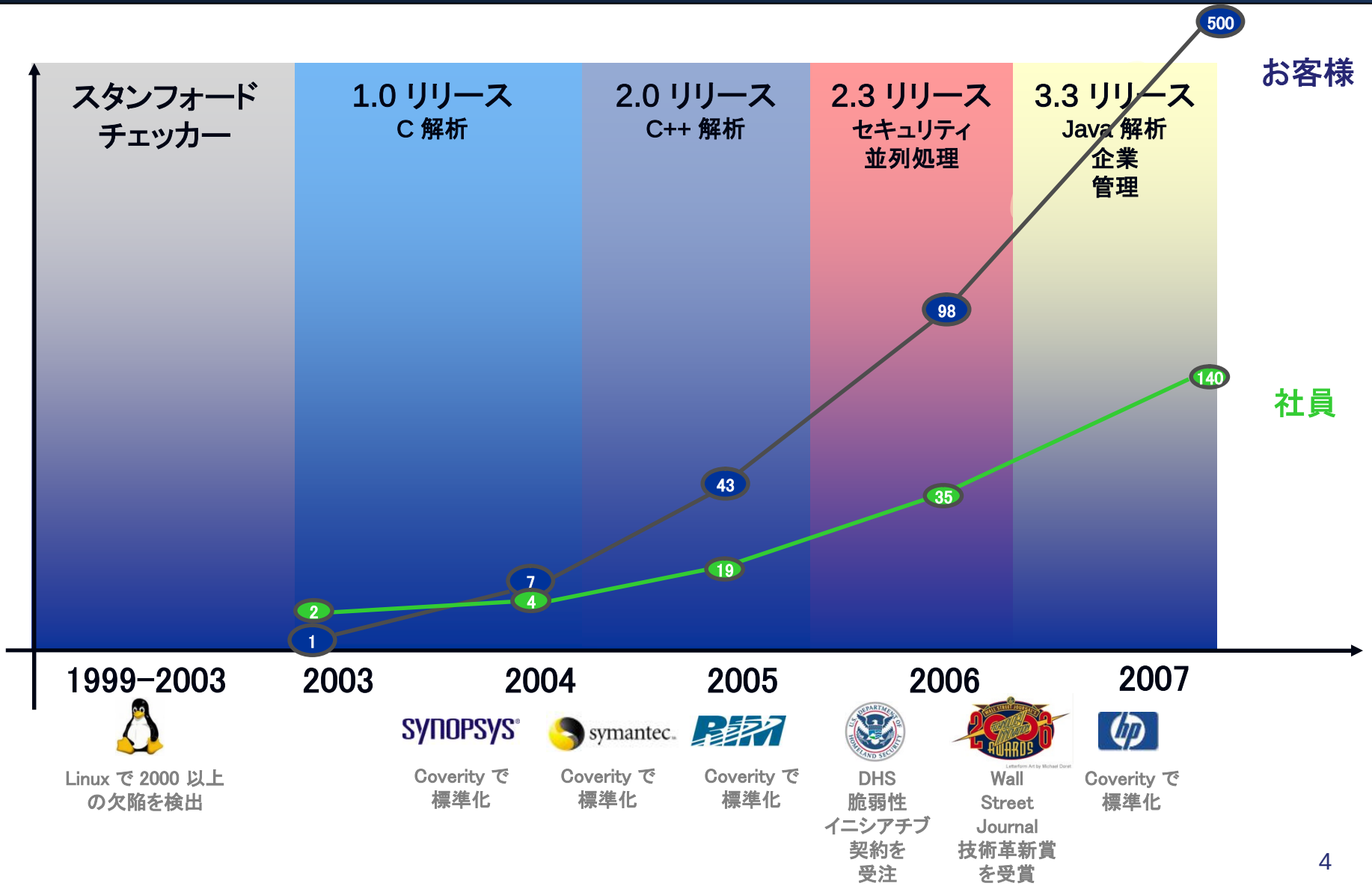
Coverityのミッション

ソース コードに含まれている重大な
バグやセキュリティ上の
脆弱性を検出すること

題目

- Coverityについて
- 開発・テストプロセスにおける静的解析の利点とPreventとArchitecture Analyzerがもたらす効果
- 静的解析ツールCoverity Preventについて
- 構造分析ツールCoverity Architecture Analyzerについて
- 投資収益率は？

当社の沿革


スタンフォード
チェッカー
1.0 リリース
C 解析
2.0 リリース
C++ 解析
2.3 リリース
セキュリティ
並列処理
3.3 リリース
Java 解析
企業
管理
お客様
社員
500
98
43
7
2
1
140
35
19
4
1999–2003
2003
2004
2005
2006
2007
Linux で 2000 以上
の欠陥を検出
Coverity で
標準化
Coverity で
標準化
Coverity で
標準化
DHS
脆弱性
イニシアチブ
契約を
受注
Wall
Street
Journal
技術革新賞
を受賞
Coverity で
標準化

製品ラインアップ

coverity

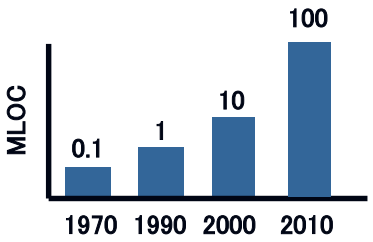
- Coverity Prevent
  - プログラムを実行することなく、データフローを静的に解析し、問題点を指摘
- Coverity Architecture Analyzer
  - 複雑なソフトウェアを視覚化して、アーキテクチャの整合性を維持
- Coverity Thread Analyzer
  - Javaにおいてのマルチスレッド環境下での並列性の問題点を検出する動的解析ツール
  - Prevent for Javaのバンドル製品
- Coverity Software Readiness Manager
  - テスト工程全体を管理するためのバグトラッキングツール

今日のソフトウェア開発における重点課題：バグの発見と修正

coverity

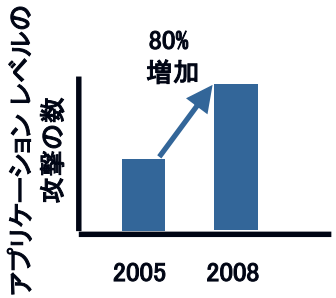
複雑化するコード	日々増大化するコードのサイズと複雑さ
高まるリスク	たった 1 つのエラーや脆弱性が顧客に大きな損害を与える
ソフトウェアのバグでリソースが浪費される	バグにより開発が遅れ、新しい機能の開発に支障が生じる

GM の車における
指数的に
肥大化するコード



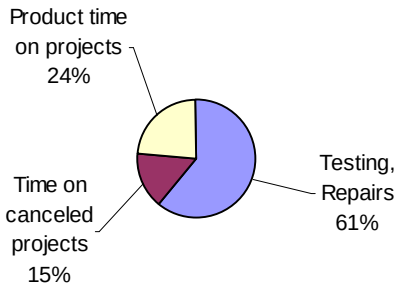
出典：Tony Scott CIO, GM

アプリケーション レベルで
増加する
セキュリティ攻撃



出典：Gartner

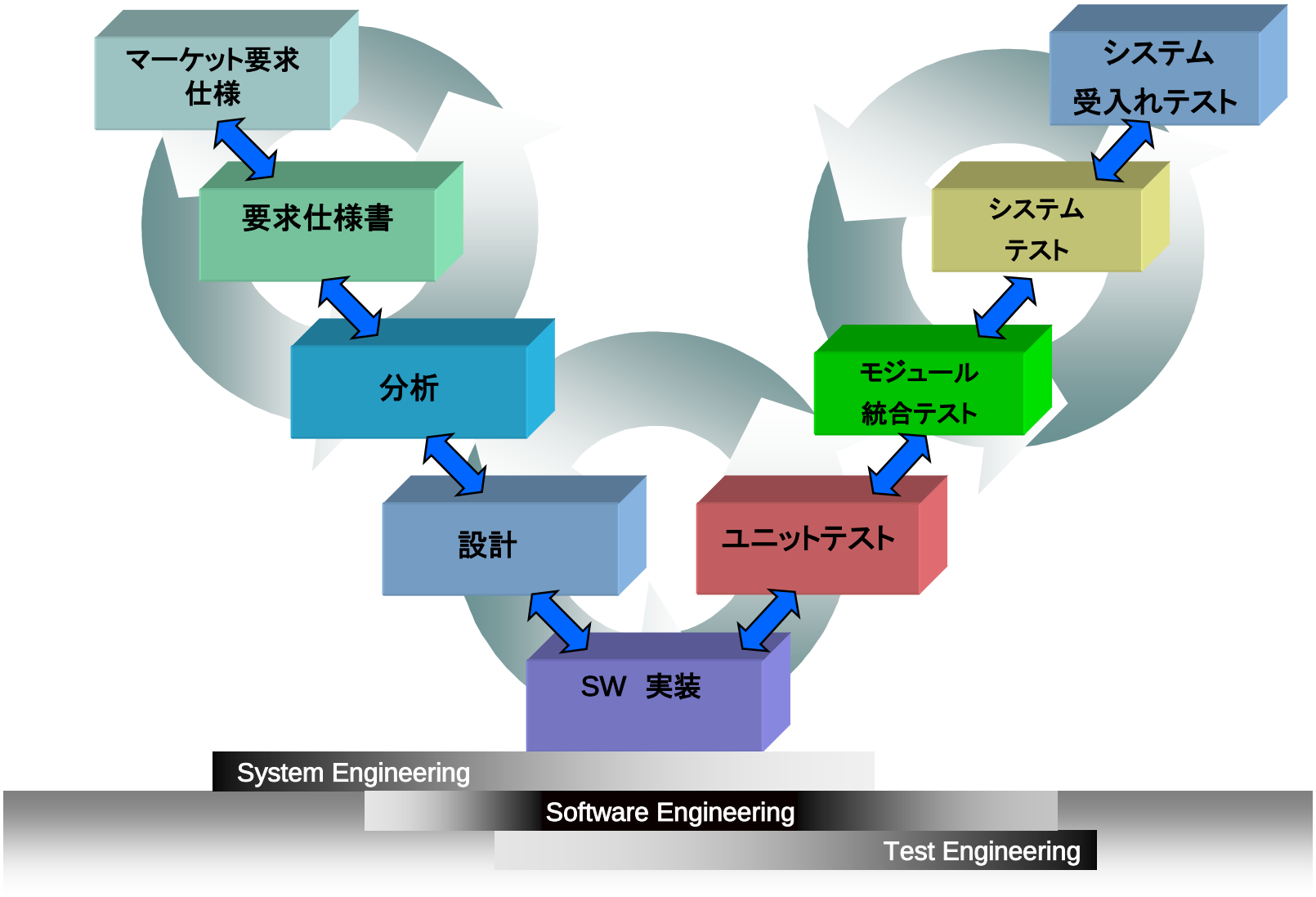
開発者が
テストとバグの修正に費やす
膨大な時間



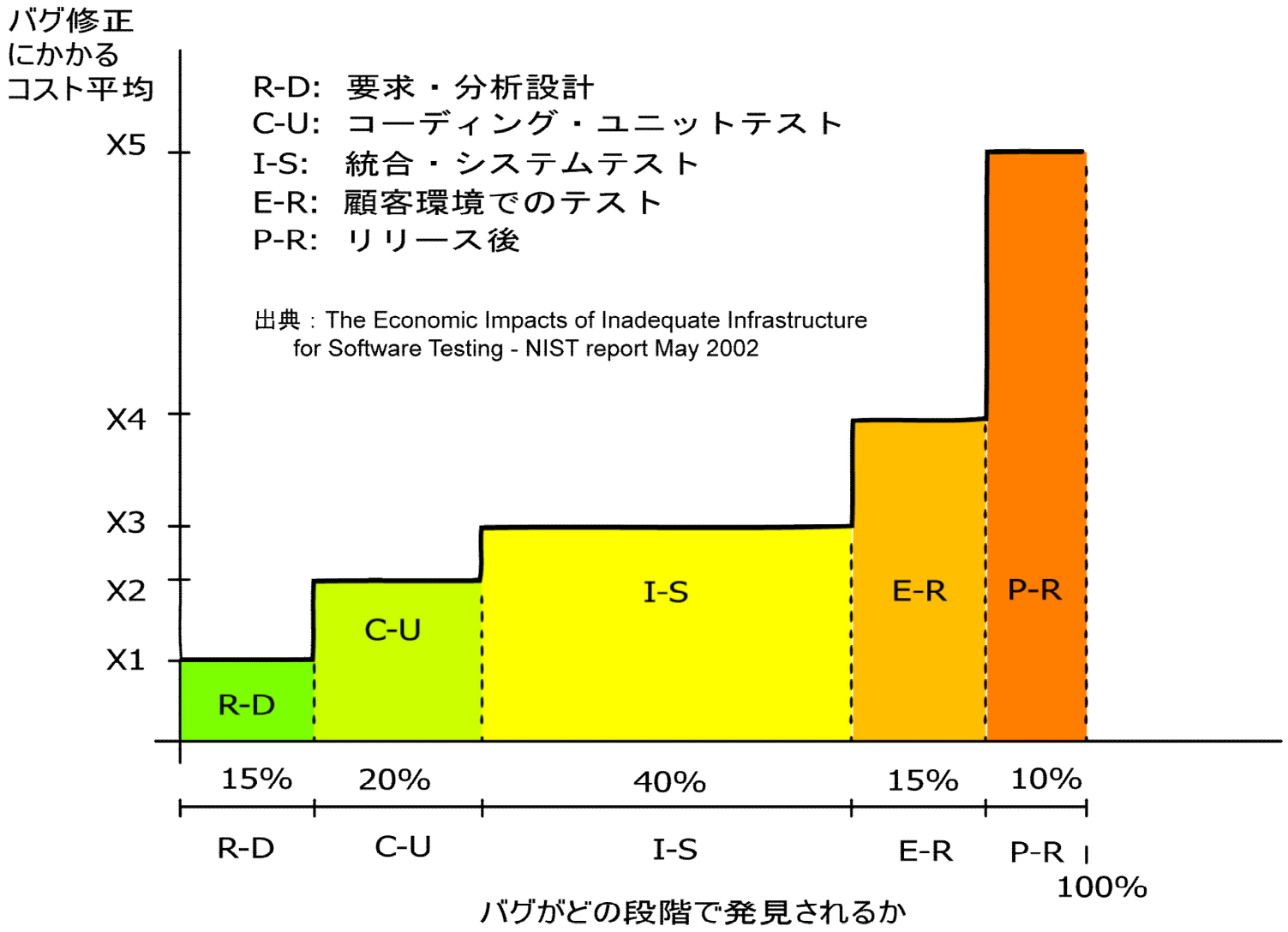
出典：Caper Jones

今日のソフトウェア開発における重点課題：バグの発見と修正

coverity

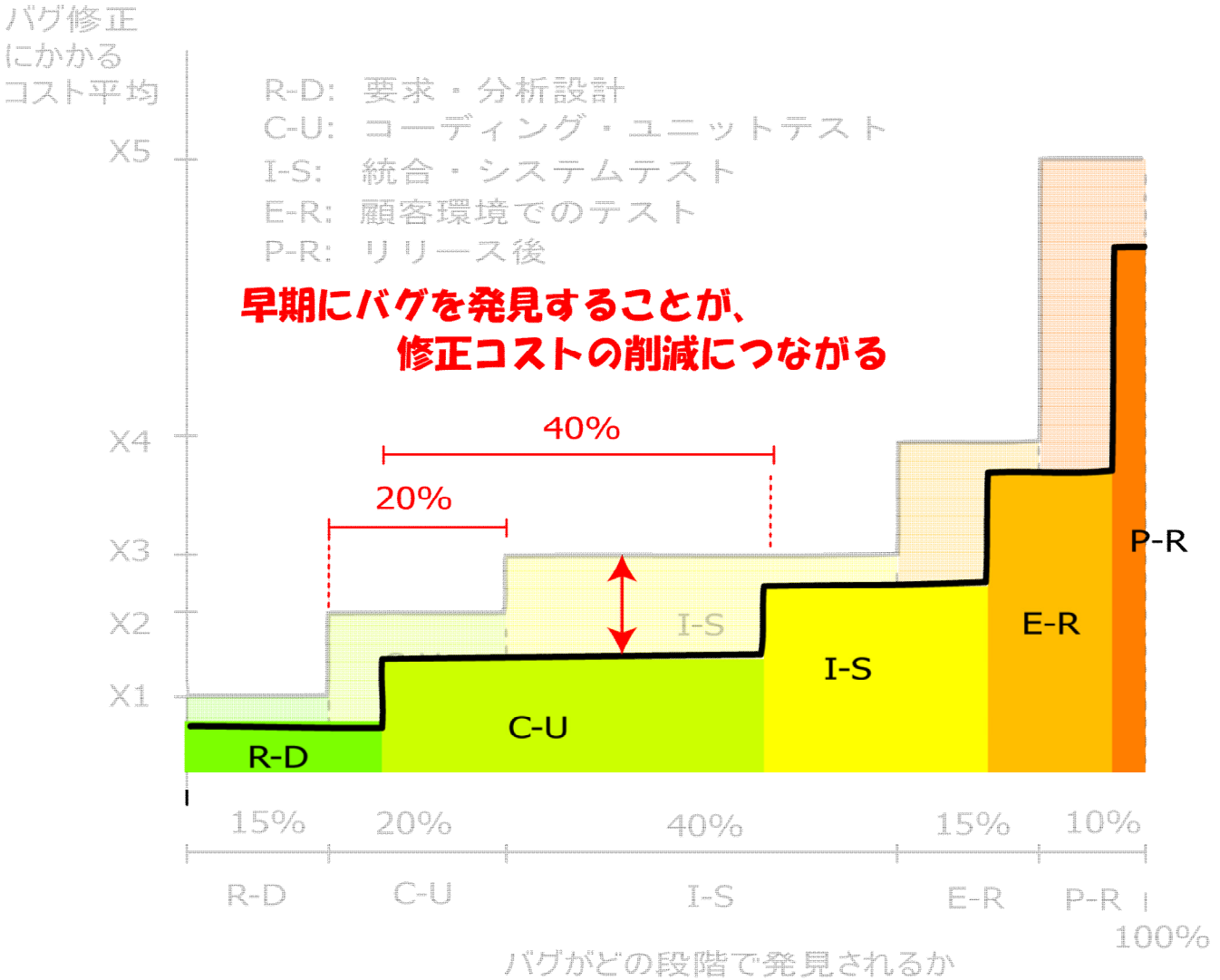


今日のソフトウェア開発における重点課題：バグの発見と修正



今日のソフトウェア開発における重点課題：バグの発見と修正

coverity



Coverityがテスト工程・手戻り工数削減に貢献！！

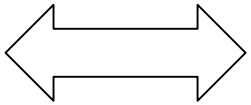
- Coverity Preventによる静的データフロー解析
  - 開発初期段階でのバグ検出
    - モジュール(関数)間にまたがった解析
    - フルパスカバレージ
    - テストケース不要
  - 単体テストの最適化
    - Preventによる静的解析+動的解析
  - プログラマーのコーディングし忘れの防止
    - 統計解析
  - 導入のしやすさ
- Coverity Architecture Analyzerによる構造の可視化
  - ソフトウェア構造が要求された仕様に合っているかどうかの検証
  - レガシーソースコードのRe-factoring

解析ツールのスタイル

coverity

動的解析

静的解析



- そのしくみ
- その強み
- その問題点

動的解析ツール

coverity

- 動的解析
  - アプローチ
    - 実行可能プログラムを実行時に観察
    - その実行プログラムで観察された違反を報告する
  - 利点
    - 動的なメモリ破壊やリソース リークといった欠陥の検出に優れている
    - 低フォールス ポジティブ（誤検知）率
    - 結果：動的解析ツールは実際の実行可能プログラムを観察

動的解析ツール

coverity

- 問題点
  - バグの発見が工程の後半
    - システムテストが中心
  - 解析時間が長い
  - カバレージに限界
    - 平均的に20～30%
  - バグ入りテスト
    - 問題の原因を不明瞭にする可能性
  - 未検出率
    - テストシナリオで実行される箇所以外は検出出来ない

静的解析ツール： Coverity Prevent

coverity

- 強み
  - 欠陥を開発過程の早い段階で検出
  - ソース コード自体に基づいた解析
  - 解析はコード全体を通して可能なパスすべてを検査
  - コードを実行する必要がない
  - テスト ケースを書く必要がない
  - スケーラブルなので、大規模で複雑なコードの解析が可能
  - プロシージャ間の解析で、漏れのない結果
  - 完全なパスカバレッジが可能

静的解析ツール： Coverity Prevent

coverity

- 問題点
  - 不完全な情報： 静的ツールは実装されていない関数と実行時のデータ値についての仮定を導入する必要がある
    - 対処法：関数モデルのサポートと作成が可能

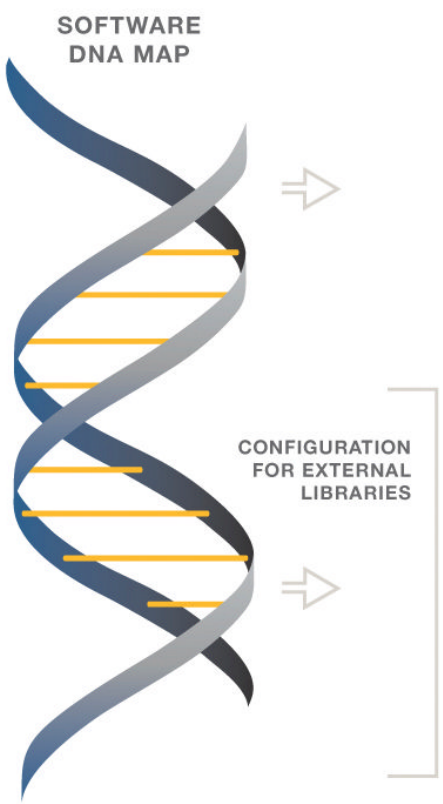
```
void my_free(void* x) {
    __coverity_free_ _(x);
}

void free(void* x) {
    /* Do nothing. */
}
```

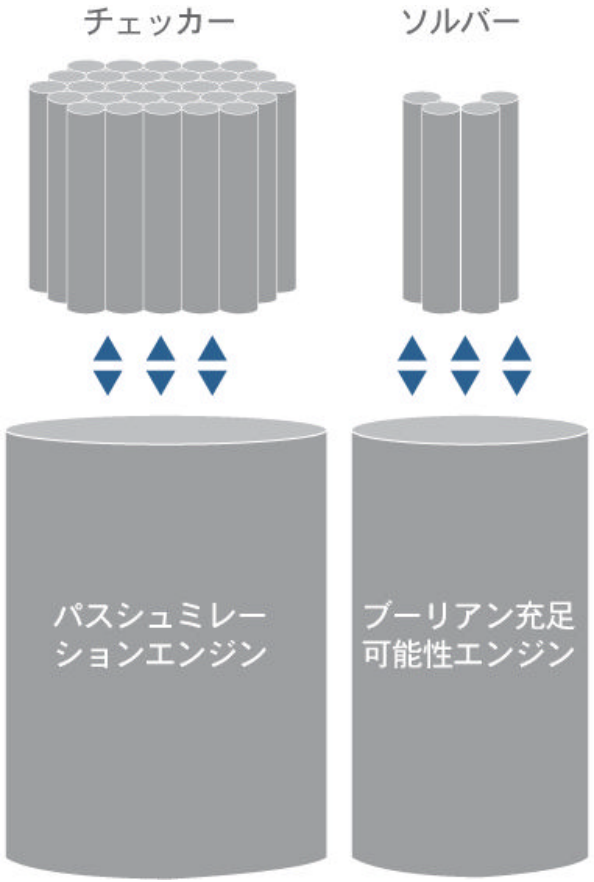
  - フォールス ポジティブ： プログラムの動作やデータ値についての誤った仮定がフォールス ポジティブにつながる
    - 対処法
      - 関数モデルのサポートとカスタムモデルの作成可
      - ブーリアンSATソルバー
      - フォールスパスプルーニング

Coverity Prevent

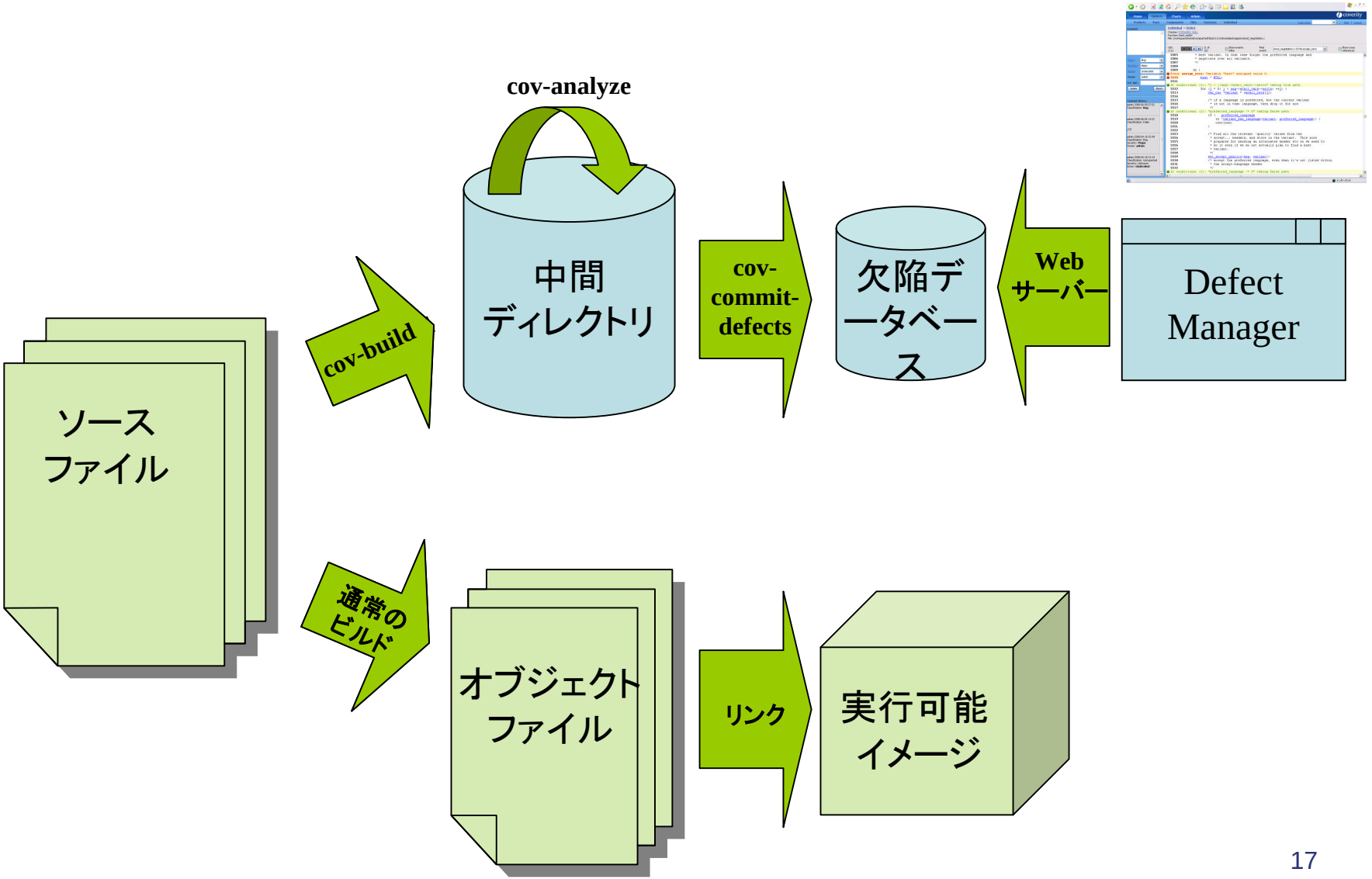
coverity

マッピング	特定	解決



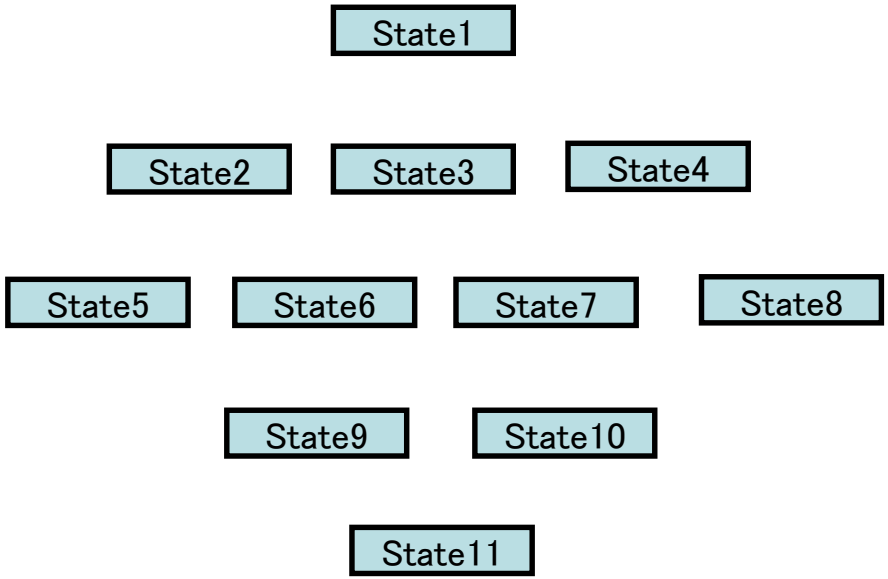


Prevent導入方法:
従来のプロセスに手を加える必要なし！



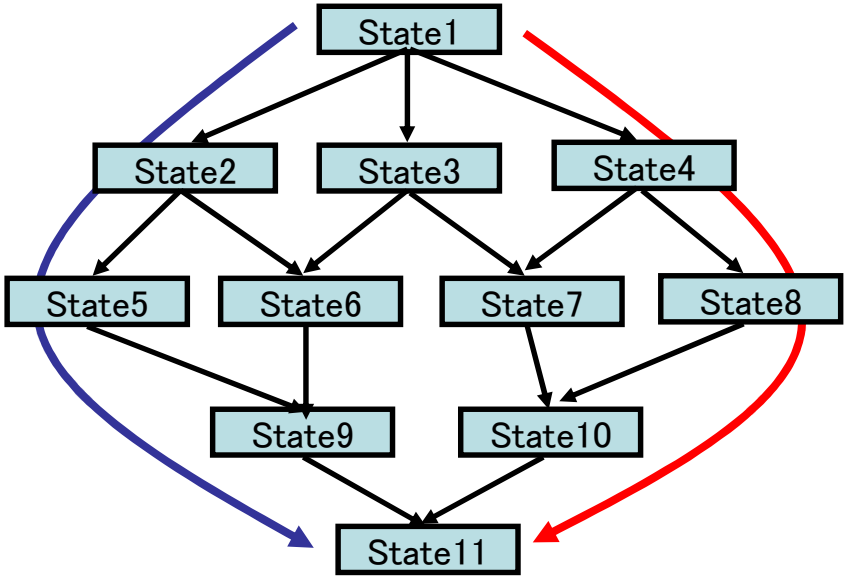
Preventによるフロー解析と静的構文解析ツールとの違い

- 一般的な構文静的解析ツールが可能なのは？



・各関数内の構文チェックが主流
・深度が浅い
・各Stateや関数間の関連はNG
・警告メッセージが多数が真に潜在するバグ検出は？
・高誤検知率

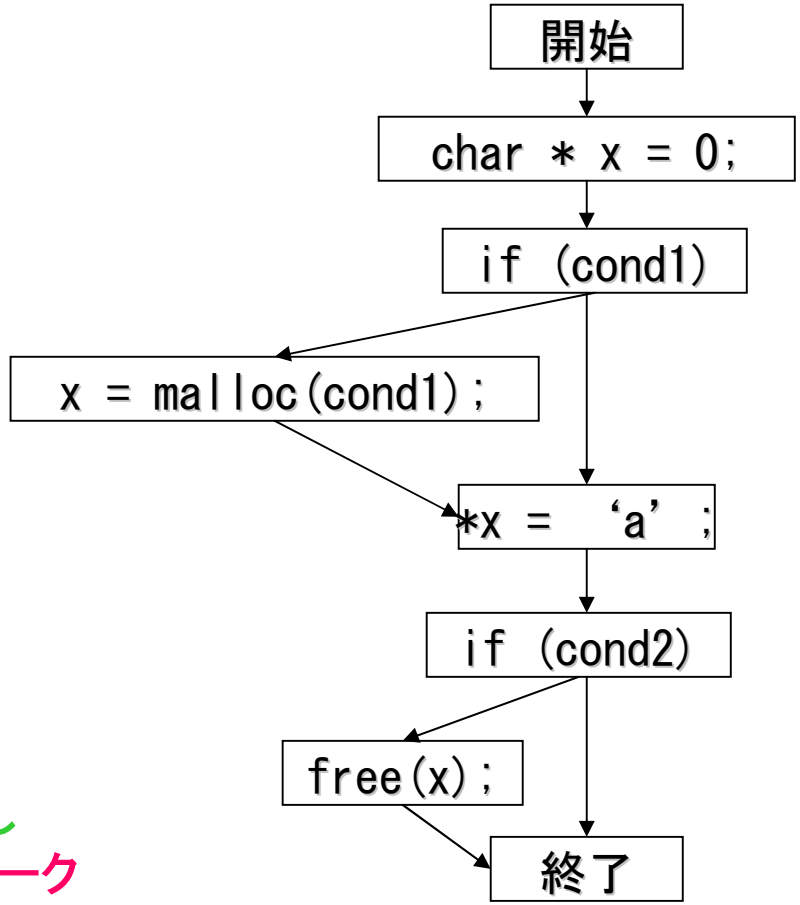
- Preventによるフロー解析が可能にすることは？



・フルパスを実行し網羅的にソースコードを解析
・各State間や関数間の依存関係を解析をサポート
・デバッグするかのようにソースコードをシミュレーション
・潜在的なバグをツールが検出
・低誤検知率

例： Preventによる静的フロー解析

```
void four_paths(int cond1, int cond2) {
  char * x = 0;
  if (cond1) {
    x = malloc(cond1);
  }
  *x = 'a';
  if (cond2) {
    free(x);
  }
}
```



four_paths(1,1); // 問題なし
four_paths(1.0); // リソース リーク
four_paths(0,1); // Null ポインタの参照解除
four_paths(0.0); // Null ポインタの参照解除

Coverity Preventのメリット

coverity

操作性	• ビルド環境＆ソースコードの改変不要
詳細な解析	• プロシージャ間解析 • あらゆるパスを解析 • 統計解析 • インクリメンタル解析
正確な解析	• <15% 誤検知率
容量	•何百万行ものコードに対応可能

インクリメンタル解析とバグ統合

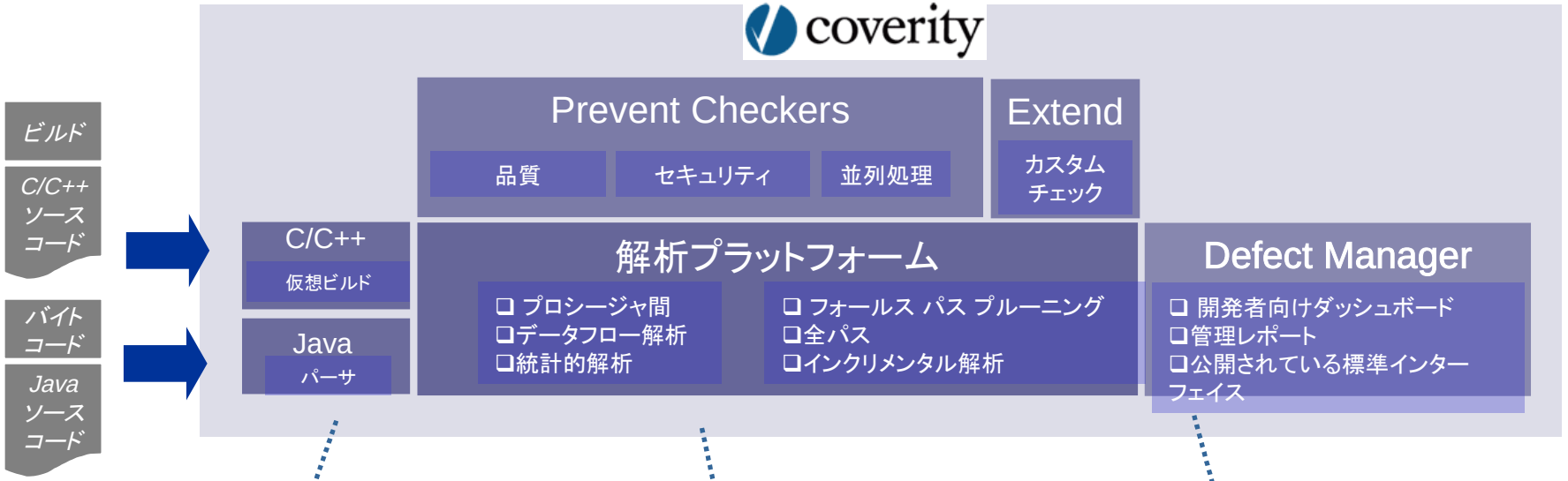
- Prevent の初回の実行後は、C/C++ ソース コードへの変更にはインクリメンタル解析のみ必要
- Iteration毎にフル解析を行わないためテスト工程の効率化
- Prevent では独自のテクノロジを使用してバグを記憶するため、ソースコードが変更されてもバグが 2度報告されることはありません。

Compare Runs Product: Samba

	Old	Common	New
CIDs:	91	106	7
Defects:	97	119 / 116	7
Runs:	10		15

Old	New ▾	Checker	Classification	Owner	Severity	Action
	221	FORWARD_NULL	False	ssorce	Unknown	Resolved
	208	UNINIT	Pending	jra	Unknown	Fix
	207	RESOURCE_LEAK	Bug	bchelf	Unknown	Fix
	205	RESOURCE_LEAK	Bug	bchelf	Unknown	Fix
	204	RESOURCE_LEAK	Bug	bchelf	Unknown	Fix
	203	RESOURCE_LEAK	Bug	jra	Unknown	Fix
	202	RESOURCE_LEAK	Bug	jra	Unknown	Fix
347	347	RESOURCE_LEAK	Bug	jmcd	Unknown	Fix
346	346	RESOURCE_LEAK	Bug	jmcd	Unknown	Fix
345	345	RESOURCE_LEAK	Bug	volker	Unknown	Fix
335	335	FORWARD_NULL	Bug	bchelf	Unknown	Fix
220	220	DEADCODE	Bug	jra	Unknown	Fix

Coverity Prevent と Coverity Extend



コードのビルド

- 自動設定
- ビルド システムの変更は不要

コードの解析

- ソース コード内の重大な実行時　バグとセキュリティ脆弱性の検出
- 高いスケーラビリティ
- コード特有のイディオムと API の解釈方法を設定

コードの修正

- 根本原因解析
- バグのライフサイクル管理
- レポート
- スクリプト言語 CLI

Prevent チェッカー ライブラリ

coverity

品質

- システムとプロセスのクラッシュ
- メモリ/リソース リーク
- データ、メモリ、ファイルの破損
- パフォーマンスの劣化
- 予期しない動作

並列処理

- デッドロック
- ロックの競合
- 予期しないパフォーマンス
- パフォーマンスの劣化

セキュリティ

- サービス拒否
- 権限昇格
- 悪質なコード

チェッカー ライブラリ

リソースの問題
- リソース リーク
  - メモリ リーク
  - ファイル ポインタ リーク
  - システム リソース リーク

並列処理の問題
- 二重ロック
- ロック解除の欠如
- 不正確なロック獲得
- ロック時のスリープ

API の使用エラー
- サイズの大きなパラメータの受け渡し
- セキュリティで保護されていない一時ファイル作成
- 不適切なメソッド オーバーライド

ポインタ エラー
- 未初期化データの使用
  - 未初期化メモリ
  - 未初期化変数

割り当て演算子の不一致

無効なポインタの間接参照
  - Null ポインタの間接参照
  - 不適切なアドレス空間
- 解放されたポインタへのアクセス
- ぶら下がりスタック参照
- 空きリソースの使用
  - 二重解放 (メモリ、ファイル ポインタ、システム リソース)
  - 解放後に使用 (メモリ、ファイル ポインタ、システム リソース)

ロジック エラー
- 欠陥のある分岐ロジック
  無効な STL イテレータの使用
- 使用不可能な演算
  一貫性のないエラー処理
- セキュリティ ロジック エラー
  - チェック時刻、使用時刻
  - セキュリティで保護されていないファイル作成
  - 不適切な chroot
  - 不適切な特権継承

セキュリティに関する警告
セキュリティを脅かす可能性のあるコーディング慣行

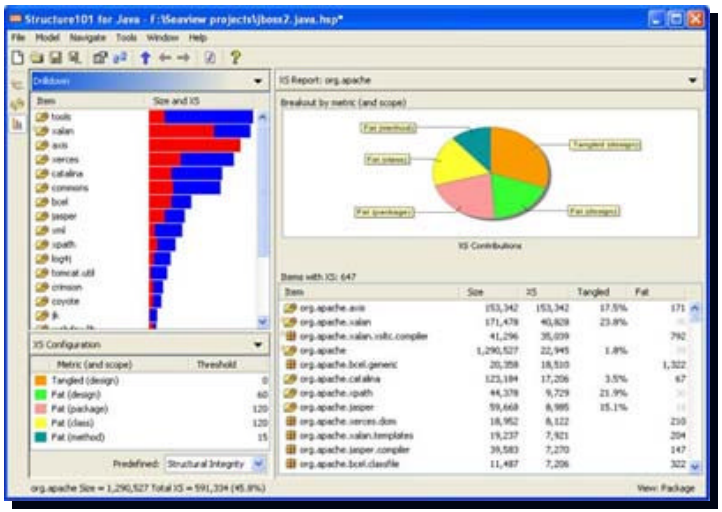
範囲エラー
- 範囲外配列アクセス
- バッファ アンダーフロー
- スタック破壊
  - スタック オーバーフロー
  - スタック バッファ オーバーラン
  - スタック文字列オーバーラン
- 負の整数への無効なキャスト
- 不適切な割り当てサイズ
- Null 以外で終了した文字列

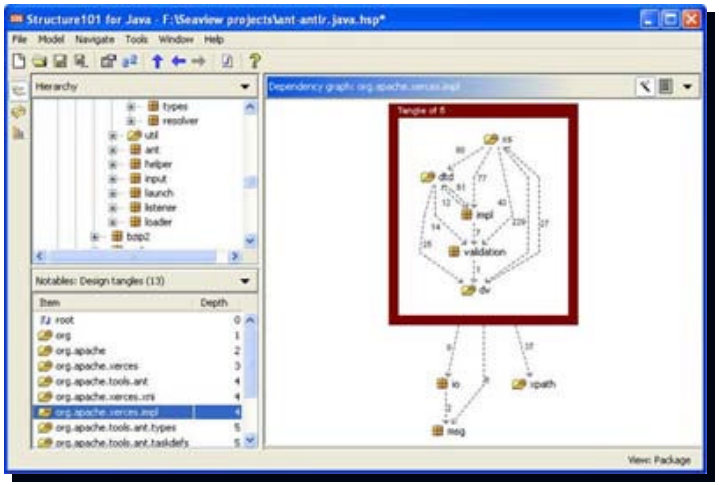
外部データ処理
- 整数
  - ループ結合
  - 配列アクセス
  - 割り当てサイズ
- 文字列
  - バッファ オーバーフロー
  - SQL インジェクション
  - 書式文字列エラー
  - クロス サイト スクリプティング

Coverity Architecture Analyzer

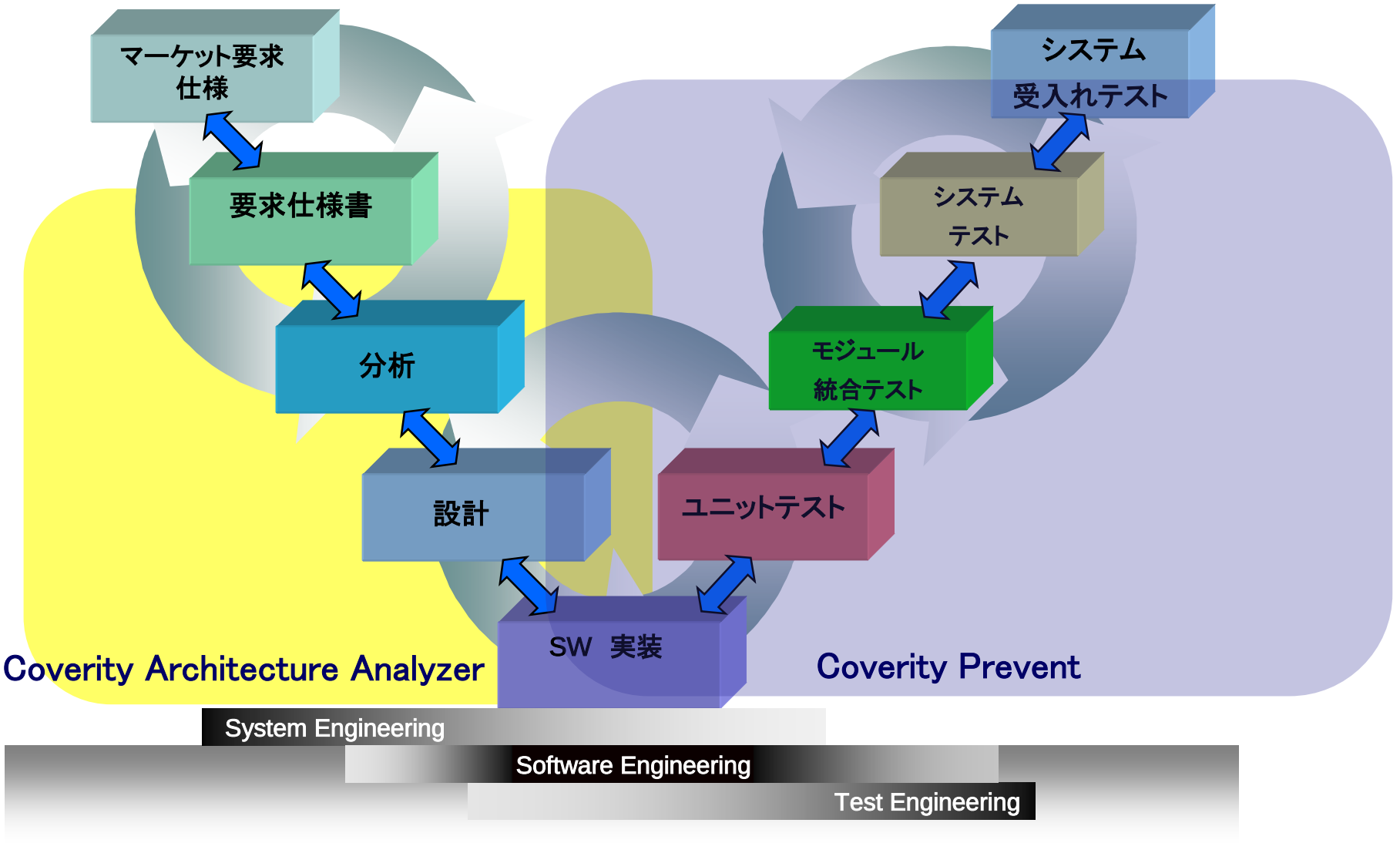
coverity

- ソフトウェア依存関係を分析
  - コール グラフを含むコード依存関係を視覚化
  - 変更およびリファクタリングの及ぼす影響を把握
  - 意図しない依存関係を除去
  - 仕様化・文書化されていないレガシーコードの構造把握
  - 要件定義と開発コードとの相関比較





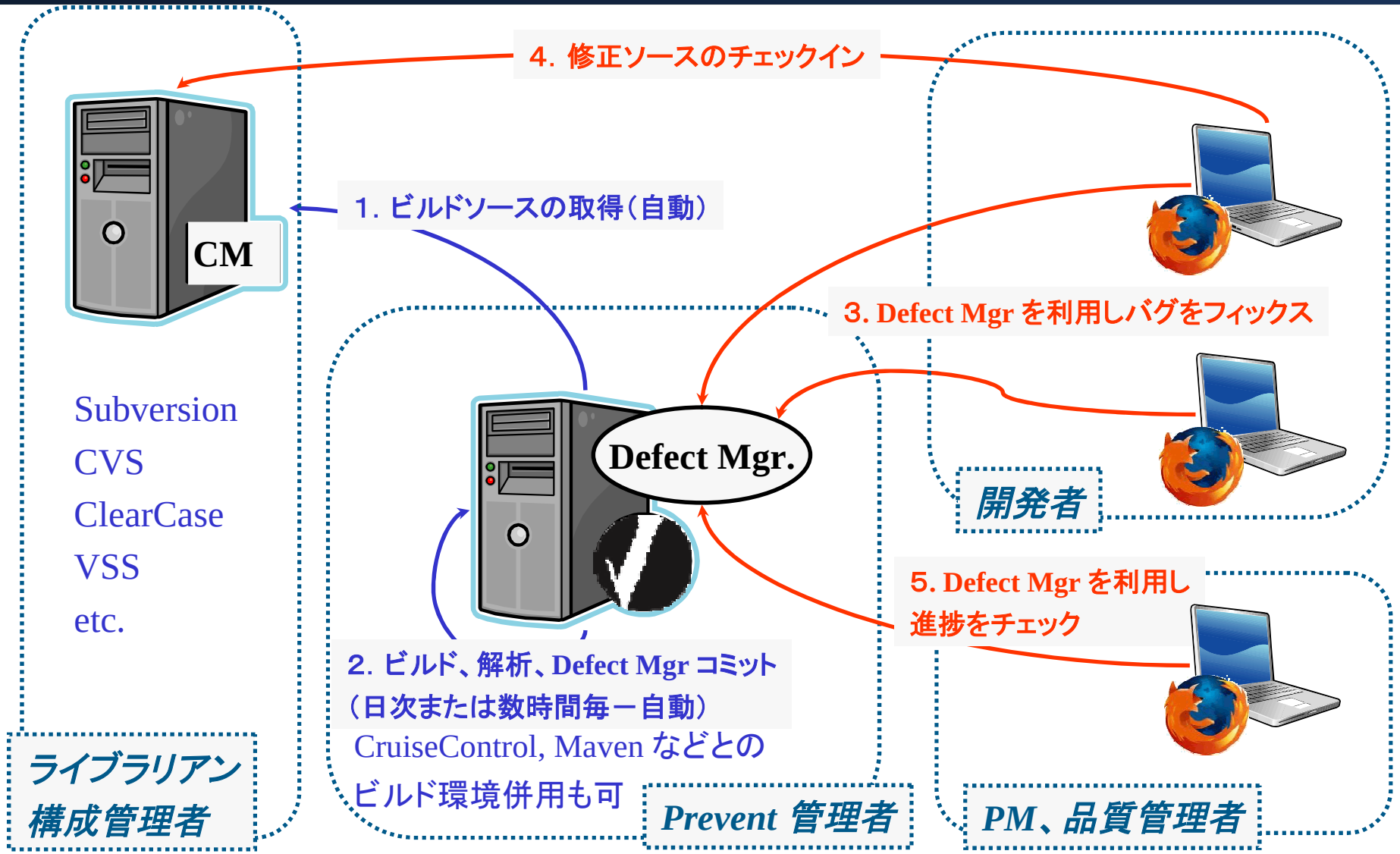
開発プロセスにおけるPreventとArchitecture Analyzerのポジション



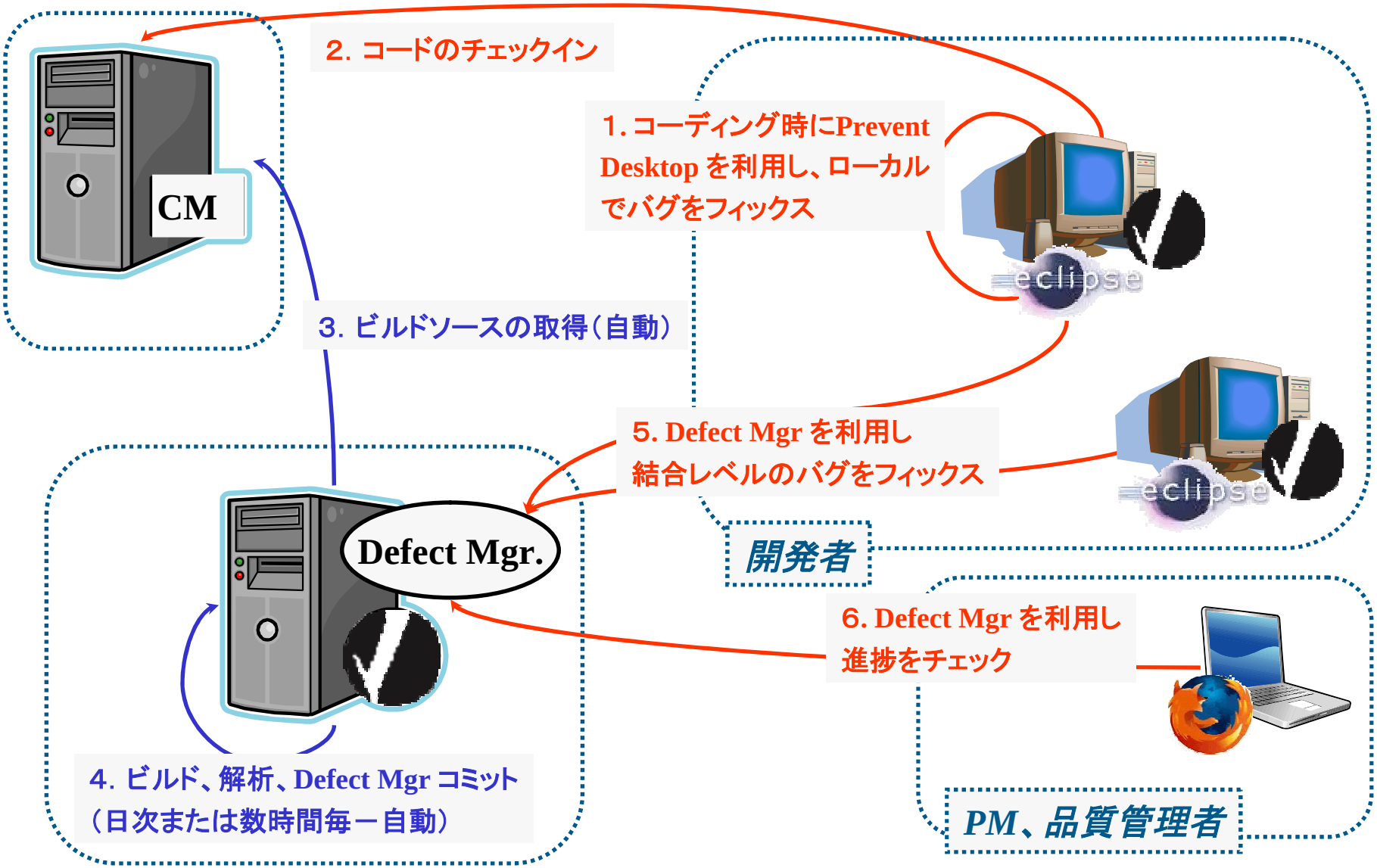
配備スタイル

- 日次ビルド(ナイトビルド)スタイル
  - Coverity 管理者による毎晩の解析実行
  - 開発者による、Prevent Defect Manager を使用した欠陥の検査
- エンタープライズスタイル
  - 上記のスタイルの機能に、開発者によるローカルマシンでの解析実行

日次ビルド(ナイトビルド)スタイル



エンタープライズスタイル



製品デモンストレーション

coverity

Coverity Prevent

Coverity Architecture Analyzer

コベリティ 日本支社
アプリケーションズ エンジニアリング マネージャー
高石　勇

ご静聴ありがとうございました

コベリティ 日本支社
〒163-0532
東京都新宿区西新宿1-26-2
新宿野村ビル32階
TEL:03-5322-2978
E-Mail: japan_sales@coverity.com
Website: http://www.coverity.com/index_jp.html