ある命令を求めたい場合には、フォワードスライシングを実行します。

図5にCodeSurferによるフォワードスライシングの結果を示します。本稿ではモノクロでわかりづらいですが、黄色でハイライト表示されている命令（❶）に対して、赤色で表示された命令（❷）がその影響を受ける可能性のあるすべての命令になります。また、この応用としてデッドコードなども識別できます。

次世代の静的解析ツールの採用事例

ここでは、「CodeSonar」を活用した静的コード解析技術の採用事例を紹介します。

医療機器向けソフトウェアの解析に関して、新技術の調査を行っている米国食品医薬品局（FDA；Food and Drug Administration）の研究者が、機器内の潜在的な欠陥を発見するためにCodeSonarを使用しました。

FDAのCDRH（医療機器・放射線保健センター）は、医療機器の市販後調査を管理監督しています。しかしながら、市販後調査を行うことは容易な仕事ではありません。最新の医療機器ソフトウェアの複雑さという条件下では、出荷されたソフトウェアの事前知識がない第三者の立場であるCDRHの調査員が調査を実施することは非常に困難で、そして時間がかかる作業です。このような場合、ソフトウェア欠陥を追跡する唯一効果的な方法は、従来、手作業でソースコード自体をレビューする方法でした。これまで、この作業を支援するために、静的な解析技術の用途を調査していました。

今回、レビュー対象の制御ソフトウェアは、主にC/C++で実装されています。そして、そのソフトウェアは、3つの独立したモジュール形式で開発され、約20万行のコードサイズで構成されています。

CodeSonarによる解析は、次のコマンドを使用し、3つのモジュールそれぞれに対して別々に実行されました（CodeSonarの解析コマンドは、通常のビルドコマンドにフックする形式になります）。

```
> codesonar hook-html project make -f project.mkfile
```

ワーニングの例

出力されたワーニングの一部を図6に示します。図中のコードは、未初期化変数ワーニングが検出されていることを示しています。

図6で示されたコードは、実際のコードの一部になります。関数S_Keyでは、いくつかのローカル変数が定義されています。ここで、変数is_

図5◎CodeSurferによる静的スライシング

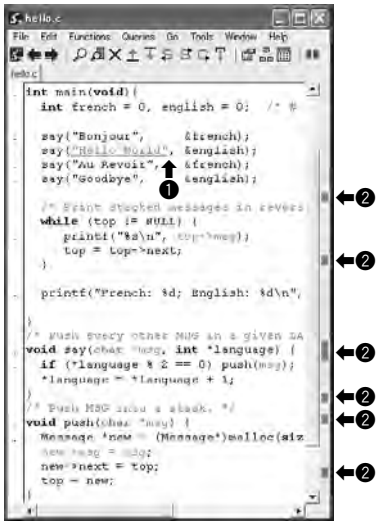

図6◎未初期化変数ワーニング

Problem	Line	Source
		someFile.c
		Enter S_Keys
true	1252	static void S_Keys (void)
	1253	{
	1254	BUTTON button;
	1255	EVENT button_press;
	1256	BOOL is_button_pressed;
	1257	unsigned int number;
	1258	
	1260	is_button_pressed = FALSE;
	1261	number = 0;
	1262	button_press = NULL;
	1263	
	1265	while (number++ < NUMBER_OF_BUTTONS)
	1266	{
	1267	if (ProcessButton ((unsigned int*)&button) == FALSE)
	1268	{
	1269	Notify (button_press);
	1270	}
	1271	
button is uninitialized	1272	switch (button) /* Uninitialized Variable (ID: 10) */
–		Preconditions for bug
		s_poqueue_number_of_elements <= 0

静的コード解析ツールの限界と進化

button_pressed、numberとbutton_pressは、関数の最初にデフォルト値で初期化されます。しかし、変数buttonは、初期化されていません。また、それは1267行目で呼び出される関数ProcessButtonで引数として渡され、そこで値を割り当ててられる可能性もあると推測することもできます。さらに、関数ProcessButtonは、条件付きループ内から呼び出されます。またProcessButtonn内でも条件付きの命令を含むかもしれません。そして、そこでは変数buttonに対して、有効な値が割り当てられない可能性もあります。そのため、1272行目のswitch命令でbuttonが使用された場合、そこが“未初期化変数”ワーニングとして出力されます。これは、switch命令に従って実行されると、思いがけないパス実行を引き起こす可能性があります。そして、それは最終的には、装置の故障を引き起こすかもしれません。

このコード解析では、上記のようなワーニングが全部で736個出力されました。ワーニングクラスごとに分類された結果を表2に示します。

表2◎CodeSonarが検出したワーニング一覧

ワーニングクラス	検出されたワーニング数	精査後の問題数
バッファオーバラン	6	
バッファアンダーラン	9	
キャストによる値変更	116	29
無視された戻り値	14	
ゼロによる割り算	1	
メモリリーク	25	
リターン命令のミス	13	1
NULLポインタ参照	62	28
冗長な条件	139	4
ビット幅を超えるシフト	2	
型オーバラン	3	
型アンダーラン	2	
未初期化変数	169	36
未到達コード	34	20
未使用値	23	
無意味な割り当て	118	9

ワーニングの精査

これらの出力されたワーニングが、今回のプロジェクト（医療機器）において、実際の問題の原因となるかどうかを判断するために、手動で精査作業が行われました。その結果、直接的に装置故障の原因ではないと判断された場合には、そのワーニングのいくつかは、このプロセス間で取り除かれました。たとえば静的コード解析では、けっして使用されない変数が割り当てられた場合、それを「未使用値ワーニング」として出力します。これらのタイプのワーニングは、一般的にそれだけでは無害であると判断される場合もあります。しかしツールでは、いくつかのセーフティクリティカルコーディング規約がそれらの使用を禁止しているので、デフォルトでは、そのワーニングを出力する仕様になっています。

この精査作業の結果として、最終的に127個のワーニングが実際の懸念事項になると判明しました。ここで、このプロセスで取り除かれた残りの609個のワーニングは、今回のプロジェクトにおいて、特に重大な欠陥ではないと判別されたということになります。そのうちのいくつかはツール側の誤検出でしたが、その多くはプロジェクトの特性上の方針であったり、文脈上から単に重要でないと判断されたものになります。

成果

この静的解析手法を利用してソフトウェアの検証にかかった全体の労力は、210人時間でした。ここには、第三者が解析作業を実施しているという事情による調整作業の時間も多く含まれます。それを考慮しても、このデータは、すべての解析を手作業のみで純粋に行った場合にかかる時間や労力と比べると、かなり少ないものであると判断されました。

そして、さらに、静的解析手法は、手作業で行うプロセスとは対照的に、ソフトウェア内のエラ

ーを正確に、そして自動的に追跡する目的においては、信頼できる手段を提供しているという結論になりました。たとえば、テストケースで実施されていないパスでさえ、精密に、かつ広範囲に解析します。そのため、一度すでに多くのテストを実施されたソフトウェアに対してでさえ、新たにバグを検出する場合もありました。

さらに、ワーニングにいたるまでのパスや条件、原因までを出力してくれるところも魅力的であると判断されました。これにより、そのワーニングが起きた過程における実行パスを追うことが容易になり、ワーニングが発生する原因のルートを追うことができます。この結果、レビュー時における時間や労力をかなり大幅に減少することに成功しました。

市場における静的解析技術のトレンドと今後の展開

これまでコーディングルールチェックのような静的解析ツールでは、ファイル間にまたがるようなメモリリークや、ゼロ割のような実行時に変数やポインタの値が決まるような動的エラーの検出は苦手とされてきました。また、メモリリークなどを動的テストツールで検出するといった方法もありますが、大きなサイズのソフトウェアでは、解析時間も長くなり、そもそも原理的に、誤検出が発生するなどの問題点もありました。

そこで最近では、今回紹介したような次世代の静的解析ツールを利用して、動的エラーを検出するという動向が増えてきていると感じています。また、ツールのバグ検出機能では対応できないような問題に関しては、「CodeSurfer」のような支援ツールを活用してレビューを効率的に行い、対策するといった傾向も見られます。そして、さらには、コーディングルールチェッカーと高精度バグ検出ツールなどを併用して、品質を二重、三重にもチェックしているような現場もあります。これは、それぞれのツールの特長や原理を活かした運用方法であるといえます。

また、機能面では解析精度や速度の向上はもとより、バグトラッキングやマネージメントシステムなどのエンタープライズ対応が強化されていく傾向にあります。今後の展開としては、次のような改良や強化が検討されています。

- バグ検出エンジンのさらなる精度向上
- 解析時間の短縮（インクリメンタル解析）
- ユーザカスタムチェック機能の拡張
- バグトラッキングシステムやマネージメントシステムの強化
- 問題（バグ）に対する視覚化
- バイナリやマシンコードの解析

おわりに

本稿では筆者の経験や所感をもとに、静的コード解析技術や静的解析ツールに関して解説してきました（一部論理的でない記述もあるかと思いますが…）。

高精度バグ検出ツールで、ファイルに間にまたがる動的エラーを検出し、さらにレビューの際には、プログラムスライス機能などを備えた支援ツールを活用するといった事例が、今後さらに増えてくると感じています。ここで説明してきた内容が読者のみなさんの品質向上活動にとって有益な情報となれば幸いです。

●参考文献

[1]『Overview of GrammaTech Static Analysis Technology』、GrammaTech, Inc.

[2]『Static Analysis of Medical Device Software using CodeSonar』、Paul Anderson/GrammaTech, Inc.

[3]『プログラムスライシング技術と応用』下村隆夫 著、共立出版 刊、ISBN4-320-02743-4