構文解析を用いたCOBOLソースコード品質点検ツールの開発

テクニカルコンピテンシー部
開発・品質技術室
アプリケーションスペシャリスト

武市　正人

Masato　Takechi
masato-takechi@exa-corp.co.jp

ソースコードの品質確保において目視点検は重要であるが、時間がかかる、見落としが出る、属人的であるため結果がばらつくなどの問題点がある。そのため近年JavaやCなどのオープン系言語では目視点検をサポートするツールが整備されてきているが、COBOLには同様のツールがない。そこでCOBOLソースコードの点検を自動化したツールCHE-COBOを開発した。本稿ではCHE-COBOの概要、自動化された目視点検ノウハウの例、またCHE-COBOの効果について報告する。

1. はじめに

ソフトウェア開発では常に高品質な成果物が求められており、品質向上のために様々な点検活動が行われている。開発の期間は限られているため、品質点検を素早く高精度に実施する必要がある。

開発下流工程の成果物であるソースコードの品質点検に目を向けると、大きくテストと目視点検の２つの方法がある。このうちテストではソフトウェアに要求される仕様を確認し、目視点検では仕様確認に加えてテストで発見しにくい誤作動、非効率な記述、保守性などの品質も確認する。このように多くの内容を確認する目視点検は品質確保の活動として重要であるが、時間がかかる割に見落としが発生しやすく、さらに属人的で精度がばらつくという問題を含んでいる。近年JavaやCなどのオープン系言語では目視点検をサポートするツールとしてFindBugs[1] やSplint[2] のようなツールが登場して、様々なプロジェクトで利用されている。しかしホスト系での開発案件が多いCOBOLについてはこういったツールが存在せず、未だ目視点検の問題を抱えている。

この状況を解決すべく、COBOLの品質点検ツールCHE-COBOを開発した。CHE-COBO開発では大きく２つの課題に取り組んだ。１つは目視点検ノウハウを明文化・定式化してツールに取り込むこと、もう１つはCOBOLソースコードをコンピュータで解析するための構文解析技術の導入である。特にツールの有効性を決定する目視点検ノウハウの取り込みについては、社内のCOBOL開発部署から直接ノウハウを提供してもらい、現場の声を取り入れる形で実現した。

さらにどれだけ危険なコードを発見できるかというツールの効果を検証するために、実際のCOBOL開発プロジェクトの協力を得て開発中のコードにCHE-COBOを適用した。その結果、致命的なバグを含む数百の危険箇所を発見しツールの有効性を確認した。

本稿ではCHE-COBOの概要と実適用事例、今後の展望について解説する。まず２章でCOBOLソースコードの品質点検活動の課題と対策について述べ、続く３章で開発したCHE-COBOについて解説する。４章では適用事例をベースにCHE-COBOの効果の解説を行い、５章で今後の展望を述べる。

2. COBOLソースコード品質点検活動の課題・対策

この章ではソースコード品質を定義し、COBOLソースコード点検活動の問題と対策を示す。

2.1. ソースコード品質の定義

ソースコードの品質点検方法に言及する前に、まずは対象とするソースコードの品質を定義する。ISO9126 / JISX0129[3] によるとソフトウェアの品質は機能性・信頼性・使用性・効率性・保守性・移植性の６つの特性に分類できる。ソースコードの品質はこれらの品質特性に直結するため、これらを使ってソースコードの品質を定義できる。ただしこれらの特性の対象はソフトウェアであるため、ソースコードの品質を説明するためには整理が必要となる。

特性の整理では、まず使用性を除外した。使用性はソフトウェア利用時の理解・習得・魅力に関する特性であるためソースコードの品質とは直接関係がない。次に品質特性の特徴を考慮して、いくつかの特性を統合した。６つの品質特性はソフトウェアの測定可能な物理的特長（属性と呼ばれる）を品質の観点から分類した品質副特性の集合として作られており、１つの属性が複数の品質副特性に関係するという特徴を持っている（図１）。つまり具体的な属性を６つの品質特性のどれか１つに分類することは難しい

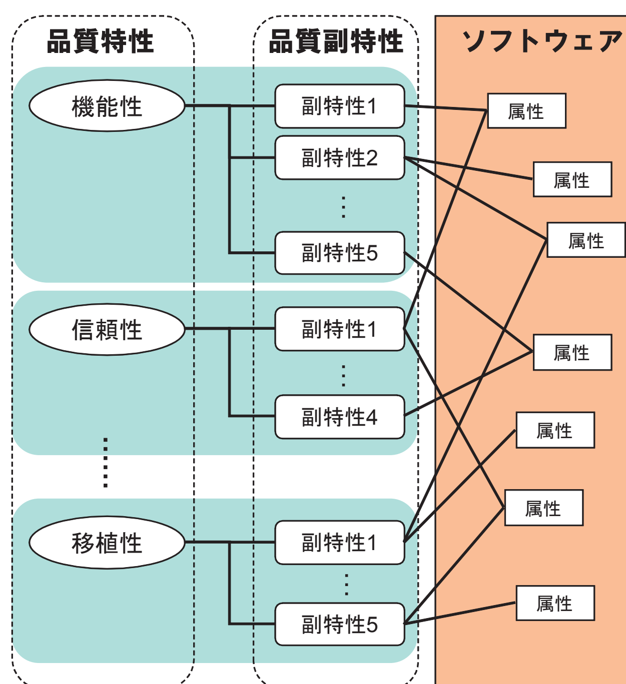

図１　ソフトウェア品質特性の構成

ということである。例えば「エラーデータを決められた方法で回復するエラー処理のコード」という属性は、機能性と信頼性の２つの品質特性と関連を持つ。属性と品質特性をつなぐ副特性は、正しいエラー処理を期待する正確性と、障害による故障の回避を期待する回復性となる。属性が２つの品質特性と関連しているため分類は不可能である。

ソースコードの品質を定義する目的が目視点検ノウハウの分類であることを考えると、品質特性を分類しやすい形に統合する必要がある。前述した「エラーデータを決められた方法で回復するコード」の例から、機能性、信頼性は動作に誤りがないことを示す特性といえる。また保守性と移植性は、大きく括ると保守しやすいかどうかを示す特性といえる。以上の観点からソースコードの品質を以下のように定義し、これらに問題のないコードを品質の良いコードとした。

●ソースコードの品質の定義

- 動作に誤りがない（機能性、信頼性）
- 効率的に動作する（効率性）
- 保守しやすい（保守性、移植性）

2.2. COBOLソースコード品質点検活動の問題と対策

2.1節で定義したソースコードの品質を点検する方法を表１にまとめた。

表１　ソースコードの品質点検活動一覧

品質の定義	点検の方法
動作に誤りがない	●単体・結合・受入などのテスト ●目視点検
効率的に動作する	●結合・システム（負荷）・受入などのテスト ●目視点検
保守しやすい	●コードフォーマットチェックツール ●目視点検

ここで注目したいのがすべての項目に登場する目視点検である。基本的な方法である目視点検は有益なのだが、次のような問題が存在する。

●目視点検の問題

- 時間がかかる
- 見落としが発生する
- チェック内容が属人的で結果の精度がばらつく

チェックリストを準備すればこれらの問題を抑制できるが、問題が完全に解決できるわけではない。そのため近年JavaやCのオープン系言語ではFindBugsやSplintといったソースコードのバグを自動チェックするツールが一般に利用されている。これらのツールには2.1節で定義したソースコード品質を網羅する多種多様なチェックルールが存在する。さらにツールによるチェックは目視点検の問題の解決に役立つため、開発に有用なツールとして多くのプロジェクトで利用されている。しかしホスト系で開発案件の多いCOBOLにはこういった機能を持ったツールが提供されておらず、上記の３つの問題を未だ内包している状態である。JavaやC言語と同様にCOBOLに対応した目視点検サポートツールを作れば、ソースコードの品質向上に役立つ。

3. 品質点検ツールCHE-COBO

この章ではCOBOLソースコードの目視点検をサポートするツールとして独自に開発した品質点検ツールCHE-COBO（Checker for COBOL Program）について解説する。

3.1. 特徴

CHE-COBOの特徴を表２にまとめた。

表２　CHE-COBOの特徴

特徴	●ソースコードを静的解析して品質点検を行う ●41のチェックルール ● チェック実行のON/OFFが可能 ● GUIによる操作
動作環境	● Java Runtime（version 5以上） ● Microsoft Excel（出力閲覧用）
対応COBOL	● IBM COBOL ベース

CHE-COBOはソースコードを静的解析して自動的に品質を点検する。本稿執筆時点で41のチェックルールを実装していて、どのルールを実行するかの設定がGUIにより簡単にできるようになっている。チェックルールは社内のCOBOL開発部署で策定されたチェックリストをベースに、

有識者へのヒアリング結果を加えて作成した。

チェックルールの具体例を表３に示す。

表３　CHE-COBOのチェックルール

動作に誤りがない	全8ルール ●型の異なるMOVEを避ける ●テーブルの件数オーバーを避ける
効率的に動作する	全1ルール ●DEAD LOGICを避ける
保守しやすい	全32ルール ●END-IFは必ず記述する ●複雑なモジュールを避ける

このうち「効率的に動作する」については現在ルールが１つしかなく、拡張が今後の課題となっている。

動作環境としてはJava Runtimeが必要である。これはCOBOL言語を構文解析するために、LGPLライセンスで提供されているライブラリ「IBM VS COBOL Ⅱ grammar version 1.0.3」[4] を利用したためである。このライブラリはJavaCC [5] というJava用の構文解析ライブラリに対応するように作成されている。

対応COBOLがIBM COBOLなのは上記の構文解析ライブラリを利用したためである。またIBM COBOLベースという表現になっているのは、IBM COBOLには多くのバージョンがあるが、CHE-COBOではバージョンの相違に興味がないため、すべてのバージョンを解析できるように構文解析を緩めているためである。

3.2.　チェックの具体例

まずソースコードチェックの流れを解説する（図２）。チェックは大きく構文解析、ルールチェック、結果出力の３つの手順で行われる。

手順１：構文解析

まず対象のソースコードを構文解析し、ソースを木構造のデータ（構文木）に変換してメモリ上に記憶する（１ソースに対して１つの構文木が生成される）。構文木内の各ノードはJavaのクラスで表現されている。

手順２：ルールチェック

手順１ですでにソースを木構造で表現しているので、それをトップから走査していく。問題のあるノードを発見した場合、問題箇所として記録する。実際には図の例のように単純に１つのノードで判断するルールだけではなく、親ノードと子ノードの組み合わせで問題と判断するルールがほとんどである。

手順３：結果出力

最後に問題をソースと対の状態でCSVに出力し、チェック完了となる。

次にチェックルールの具体的な処理の例として、表３で示したルールのうち『END-IFは必ず記述する』、『テー

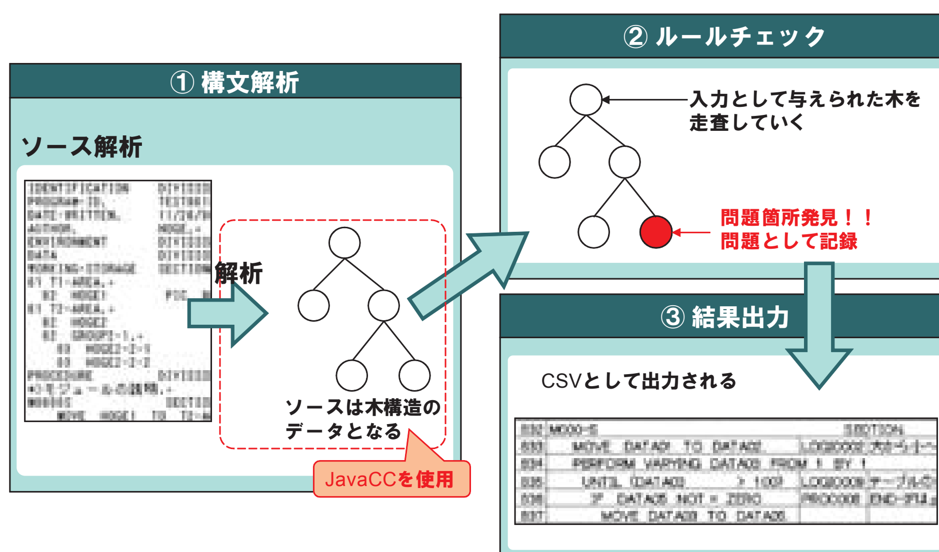

図２　ソースコードチェックの流れ

ブルの件数オーバーを避ける』について解説する。

１）END-IFは必ず記述する

このルールはEND-IFの記述漏れを検出する。 COBOLではEND-IFを記述しなくてもピリオドによりIF文の終端を記述できるが、ピリオドは見落としやすいためにメンテナンス担当者の誤読、誤修正の要因の１つになっていた。そのためEND-IFの存在をチェックするルールを組み込んでいる。例えば図３のコードを検査することを考える。

```
1:IF  DATA01   = ZERO
2:    (処理1)
3:ELSE
4:    (処理2)
5:END-IF.
```

図３　END-IFのサンプルコード

このコードを構文木にした場合、図４のようになる。END-IFが記述されていない場合、END-IFノードのない構文木が生成されるので、ルールのチェックが可能となる。

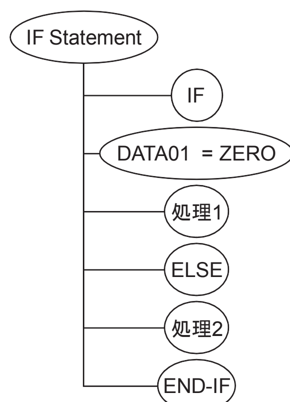

図４　IF文の構文木

２）テーブルの件数オーバーを避ける

このルールはOCCURSで定義されたテーブル件数を超えてアクセスが行われるケースを検出するルールである。例えば図５のようなコードを検査することを考える。このコードの問題は100と宣言されたテーブル件数(4行目)を超える値の添え字でテーブル項目の配列にアクセスしているループ内にテーブル項目が入ってしまっている部分(8,9行目)である。

このチェックルールの検査手順を以下に示す。

手順１：テーブル使用箇所の探索

構文木を走査し、PROCEDURE DIVISION内にある()付きのデータ項目を記録する。このとき()内の添え字も記録する（データ項目の部分参照をしていても()付きとなるが、対象でないため除外する）。このソースの場合、DATA01、COUNTERが記録される。

手順２：テーブル定義の探索

DATA DIVISION部分の構文木を走査し、DATA01の定義を探索する。DATA01があればDATA01自身、もしくはDATA01が所属する集団項目にOCCURSが宣言されていないかチェックし、宣言されていればその数値とデータ項目名を対にして記録する。このソースの場合、DATA01が所属する集団項目TABLE01にOCCURSが宣言されているため、その数値であるDATA01と100を対にして記録する。

手順３：条件式の探索

PROCEDURE DIVISION内にあるDATA01からさかのぼる形で構文木を走査(DATA01の親、そのまた親と走査する)し、さらに親の持つ子ノードを走査して条件文を記述している箇所を探索する。条件文はCONDITIONという名称のノードで構文木に登場し、PERFORM、IF、EVALUATEなど条件文を含む記述すべてが子ノードとしてCONDITIONノードを持つ。そのため親を走査していくとそのノードが所属する条件文を見つけることができる(図６)。このソースの場合、PERFORMの子ノードにあるCONDITIONが見つかる。

手順４：定義と条件式の比較

条件文の構文木を走査し、手順1で記録した()内の添え字が利用される箇所を探すと「COUNTER > 200」が見つかる。ここで手順1,2での記録内容を見るとCOUNTERを添え字として使用していたDATA01のOCCURS数が100であることがわかる。つまりこの条件式ではCOUNTERは101を超えてはいけないはずである。しかしこの条件式ではCOUNTERは200まで進んでしまい、DATA01が所属するテーブルの限界数を超えてしまう。以上の手順を持ってこの条件式の記述を誤りと判断し、CHE-COBOはこの行を問題として報告する。

4. CHE-COBOの効果検証

開発したCHE-COBOが実際にソースコードの問題点を指摘でき、品質向上に貢献できるかどうか確認するため、

```
 1: DATA                                            DIVISION.
 2: WORKING-STRAGE                                  SECTION.
 3: 01 TABLE-AREA.
 4:     02   TABLE01    OCCURS 100.
 5:             03   DATA01    PIC X(05).
 6: PROCEDURE                                       DIVISION.
 7:     PERFORM   VARYING  COUNTER   FROM  1  BY  1
 8:         UNTIL   COUNTER > 200
 9:                 MOVE   COUNTER   TO   DATA01(COUNTER)
10:     END-PERFORM.
```

図５　END-IFのサンプルコードテーブル件数オーバーのサンプルコード

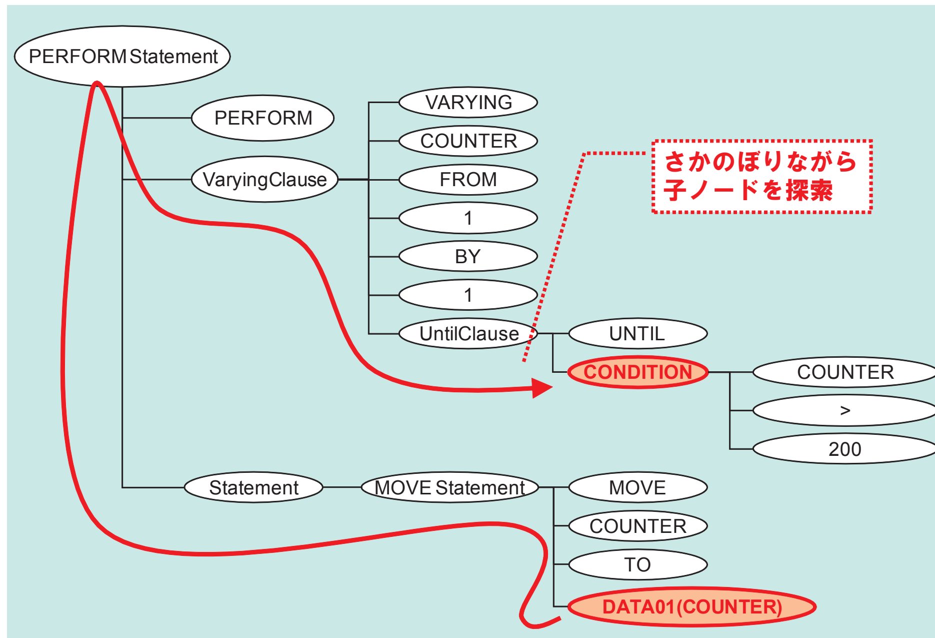

図６　CONDITIONノードの探索

社内のCOBOL利用部署の協力を得て実際のプロジェクトへの適用を実施した。この章では適用プロジェクトの概要と適用の結果を解説する。

4.1.　実適用概要

CHE-COBOの効果検証のため、以下の規模のプロジェクトに適用した。

- COBOLソースコード数：81本
- 総ステップ数：約60,000行

適用は未レビューのものを含む開発中のソースコードに対して実施し、どの程度問題を発見できるかを観察した。

4.2.　適用結果と考察

適用の結果発見された問題数は全部で5804件であった。その内訳を表４に示す。表４には表３で挙げた一部のルールについても件数を記載した。

表４　CHE-COBOの適用結果

	指摘件数
動作に誤りがない	全2169件 ●型の異なるMOVEを避ける（1947件） ●テーブルの件数オーバーを避ける（13件）
効率的に動作する	全1件 ●DEAD LOGICを避ける（1件）
保守しやすい	全3634件 ●END-IFは必ず記述する（6件） ●複雑なモジュールを避ける（112件）

多数の指摘があったが、そのうち40%が“動作に誤りがない”に関わる項目、60%が“保守しやすい”に関わる項目であった。ルールが１つしかない“効率的に動作する”については１件のみの指摘であった。全体として５分程度のわずかなチェック時間で多数の有益な指摘が見つかった。しかも41のルールについてはツールによる厳格なチェックであるため、見落としや精度のばらつきが発生せず、2.2

節で挙げた目視点検の問題が解決できている。
次に品質定義の各項目について考察する。

１）動作に誤りがない
２つの例のうち『型の異なるMOVEを避ける』の指摘件数が非常に多くなっているが、理由はこのルールの仕様にある。このルールはサイズの異なるデータ項目間でのMOVE（サイズが大きい項目から小さい項目へのMOVE、またその逆）と型の異なるデータ項目間でのMOVE（文字項目と数値項目間でのMOVE）の両方をチェック対象に含んでいる。サイズが小さい項目から大きい項目へのMOVEは一概にすべてが問題とはいえないのだが、この記述が多数発見されたため指摘件数が多くなった。このチェックルールに関しては今後一考の余地がある。とはいえサイズの大きい項目から小さい項目へのMOVEも複数検出されており、定義ミスの可能性のあるコードを発見できている。
また『テーブルの件数オーバーを避ける』は目視点検では非常に発見しづらいバグそのものであるため、ツールの有効性が確認できた

２）効率的に動作する
ルールが１つしかないため指摘件数が１件のみであったが、わずかな時間で60000行のソースコードの中から１箇所のみの問題点を発見できた点については、ツールの利用効果を実証できた。

３）保守しやすい
例示した２つのルールの指摘件数は少ないが、全体としては多数の指摘があった。例以外のルール、特にフォーマットチェックを行うルールに指摘が多かったため、指摘数が多くなっている。これはレビューされていないものを含む開発中のソースコードに対してツールによる厳格なチェックをかけたことが原因と考えられる。
『複雑なモジュールを避ける』はある一定以上の循環的複雑度であれば指摘を行うルールであり、112のモジュールが複雑であると判定された。複雑であればあるほどテストや保守が困難で不具合が入り込みやすくなるため、このルールで指摘された112のモジュールについては修正かテストによる細かなチェックが必要となる。チェック結果を追えばどのモジュールが複雑であるかが簡単にわかるため、問題部分の探索に役立つ。

以上いくつかの課題を残してはいるが、多数の有益な指摘ができたことにより、CHE-COBOがCOBOLソースコードの品質向上に役立つことが確認できた。

5. 今後の展望

実プロジェクトへの適用で効果が確認できたCHE-COBOであるが、さらに有用なツールとするためにいくつかの課題がある。一方、構文解析技術を使用した別種のツールの展望も得られた。この章ではCHE-COBOの今後の課題と構文解析技術の応用について解説する。

5.1. CHE-COBOの今後の課題

CHE-COBOをさらに良いツールにしていくために、大きく４つの課題がある。

１）新機能・ルール追加
CHE-COBO利用者数が増えるにつれ、様々な新機能、新ルール実装の要望が寄せられている。代表的なものとして、実行結果のサマリ機能やサブプログラムへのパラメタ受け渡しチェック機能の追加、未使用データ項目をチェックするルールの追加などが挙げられている。また要望として挙げられたもの以外にも、社内にはまだまだ有益なノウハウが存在するはずなので、見えている要望を組み込みながら見えないノウハウを掘り出していく必要がある。特に3.1節で述べたとおり“効率的に動作する”についてのルールは重点的に充実させていかなくてはならない。

２）偽陽性の除去
4.2節で述べたとおり、CHE-COBOの指摘項目にははっきりと問題であると言い切れないものが存在する。これは偽陽性（false positive）といわれる問題で、ウィルスチェックソフトによるウィルス誤検出問題と同質の問題である。この問題が多発した場合、開発者はツールの出力を無視するようになり、ツールが使われなくなってしまう。今後継続的に既存チェックルールの見直しを行い、偽陽性を除去していく必要がある。

３）IBM COBOL以外のCOBOL対応
CHE-COBOはIBM COBOLをベースとしており、それ以外のCOBOLには未対応である。しかし他のCOBOLに対応できれば、さらに有用なツールとなることは間違いない。特に社内でも利用されているMicro Focus COBOL（以降

MF-COBOL）については対応希望の声が大きい。以前400本ほどのMF-COBOLのソースコードに対してCHE-COBOを実行する機会があったが、80%程度のソースコードには適用が可能であった。今後解析成功率を高めていきたい。

4 ）実行準備の手間の削減

CHE-COBOの利用全体像を図７に示した。CHE-COBOはホスト上での動作を想定していない。このため実行前の準備としてホストからCHE-COBO配置PCにチェック対象のソースコードとCOPY句で参照されるソースコードの両方をダウンロードする必要がある。COPY句参照ソースコードは大量にあるため、作業にはかなりの手間がかかってしまう。この課題についてはCHE-COBO側の機能を充実させるだけでなく、CHE-COBO利用者と連携して運用方法を整備する必要がある。社内のCHE-COBO利用部署には、日時でホストからCOPY句をダウンロードする共有サーバーを準備することでこの課題への対応を検討している部署がある。CHE-COBO実行マシンからは共有サーバーを参照し、実行マシン上に配置した点検対象ソースコードをチェックする運用である。本稿執筆時点ではこの運用は準備中であるが、実行準備の手間を軽減できると予想される。今後こういった運用方法を整理し、CHE-COBO利用を促進していきたい。

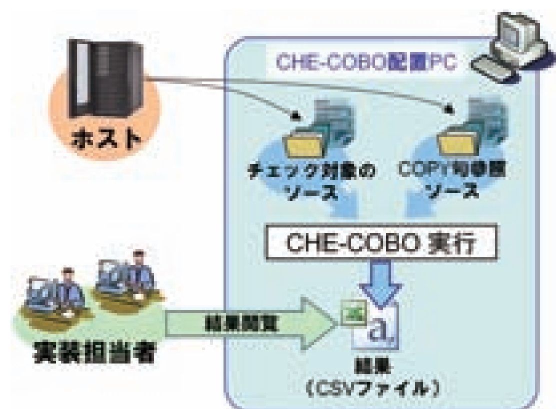

図７　CHE-COBO利用全体像

5.2. 構文解析技術の応用

構文解析の最大のメリットは、ソースコードに書かれた定義やロジックをコンピュータプログラムで把握し、操作できることである。構文解析を応用すればCHE-COBOのようにソースコードから様々な情報を収集することも可能であるし、目的に応じてソースコードの書き換えやソースコードそのものの作成も可能である。例えば次のようなツールが考えられる。

- コピー＆ペーストで作られたコードを発見するツール
- ソースコード内の分岐やサブルーチン呼び出しを図に書き出すプログラム構造可視化ツール
- COBOLで書かれたソースコードをJavaにコンバートするツール

こういったツールを作成して開発現場で利用すれば、低コストで品質向上や成果物作成が可能となる。

6. おわりに

本稿では一般的な目視点検の問題について触れ、COBOL言語における問題対策が他言語に比べて不足していることを指摘した。そしてCOBOL言語に提供されていなかった目視点検サポートツールとして開発したCHE-COBOについて解説し、その効果を示した。CHE-COBOはまだまだ発展途上ではあるが、2.2節で挙げた目視点検の問題を解決できるという点で有効なツールである。今後さらに適用数を増やし、フィードバックを得ることでツール自体を育てていく所存である。

参考文献

1) FindBugs,
http://findbugs.sourceforge.net/
2) Splint,
http://www.splint.org/
3) ISO9126 / JISX0129
http://www.jisc.go.jp/app/pager?id=42935
4) VS COBOL II grammar version 1.0.3,
http://mapage.noos.fr/~bpinon/a_cobol_parser.htm
5) JavaCC,
https://javacc.dev.java.net/

Javaに関するすべての商標は、米国Sun Microsystems, Inc.の米国およびその他の国における商標または登録商標です。
FindBugsは、The University of Marylandの商標です。
Micro Focus COBOLは、Micro Focus Limitedの商標です。
その他の会社名、製品名およびサービスは、それぞれ各社の商標または登録商標です。