89

要因組み合わせによる大量のテスト項目実施における障害の早期検出および工数削減の取り組み[1]

1. 概要

本編では、テスト項目自動生成による大量テスト実施での問題を解決する手法を提案し、適用した事例を紹介する。

通常、ソフトウェアテストにおいては、テスト要因を分析し、その要因を組み合わせてテスト項目を作成し実行する。要因組合せにおいてはペアワイズ法などを用いて少ないテスト項目で効果的にテストを実施するが、それでも要因表が巨大になると項目が膨大になる。実行と結果確認を自動化しても、現実的な時間でそれらのテスト項目を実施できず、実施可能な組合せ数に限界があった。そこで我々は、要因をランダムに組み合わせ大量のテスト項目を自動生成し、生成されたテストの実行および回答との結果比較まで自動化する手法を提案した。これにより組み合わせのバリエーションを拡張したテストを実施し、多くの障害を検出した。

しかし、活用の実績から 2 つの問題があることが明らかになっている。

1. テスト項目を、ランダムに要因を組み合わせ生成した順に実行するが、障害を早期検出するための品質リスクに応じた順番でテストできず、早期の品質確保ができない
2. NG となるテスト項目を大量に検出した場合、それらが同一の障害原因により発生するものかどうかの確認は人手であり、工数がかかる

本編では、これらの問題を解決するため、障害に関係する重要な要因を含むテスト項目を優先的に実施する手法と、検出した障害を、そのテスト項目の要因に着目して同一原因かどうか自動で分類し、原因の究明を効率化する手法を提案し、実際のテスト作業においてこの手法を適用した効果について紹介する。

2. 取り組みの背景と目的

ソフトウェアテストにおけるテスト要因表からの組み合わせによるテスト手法として、ペアワイズ法がよく知られている。この手法は、統計的に障害の原因となるのは 2 因子、3 因子の組合せであり、全ての組合せを網羅せずとも効率的なテストが可能という考えに基づく。一例では図 89-1 に示す通り、全網羅のために約 6 万件のテストが必要な要因表に対して、3 因子網羅率に着目しテスト項目を 25 件に削減できる。

---

[1] 事例提供：富士通株式会社　百足 勇人氏

しかし、例えばプログラミング言語のコンパイラでは、ユーザは既存の言語仕様全てを自由かつ複雑に組み合わせてプログラムを記述し、コンパイラに入力して翻訳する。このコンパイラの言語仕様テストにおいては、多くの要因が相互に影響し、4 因子以上の条件でバグが発生するケースも多く 2・3 因子網羅では不十分である。一例では図 89-2 に示す通り、単純な 10 行のテストプログラム 1 本にさえ 10^31 通りの要因組み合わせパターンがあり、既存のテスト手法では絞り込みが難しく、現実的な時間でテストできない。

因子数	水準数	全網羅　テスト数	削減後　テスト数※
5	2	32	6
10	2	1024	7
10	3	59049	25

※自社製組合せテストツールによる2因子網羅率100%, 3因子網羅率70%の値

図 89-1 要因表の大きさとペアワイズ法による組合せテスト項目数

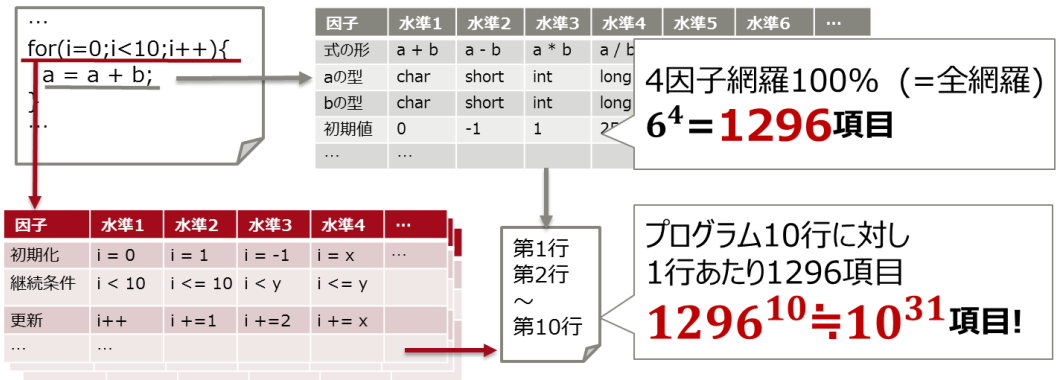

図 89-2 コンパイラの言語仕様テストにおけるテスト項目数

そのため、我々は疑似乱数を用いランダムに要因を組み合わせ、テストプログラムを生成し、大量テストを実施可能とするツールを開発した。本ツールの概要を図 89-3 に示す。本ツールは、テストプログラムの生成に限らず実行および判定も繰り返し自動で行う。これを活用したコンパイラのテスト実施により、多くのブロック構文および式を記載した巨大な要因表を基にテスト実施が可能となり、飛躍的に組合せのバリエーションが拡張され、多くの障害を検出することができた。

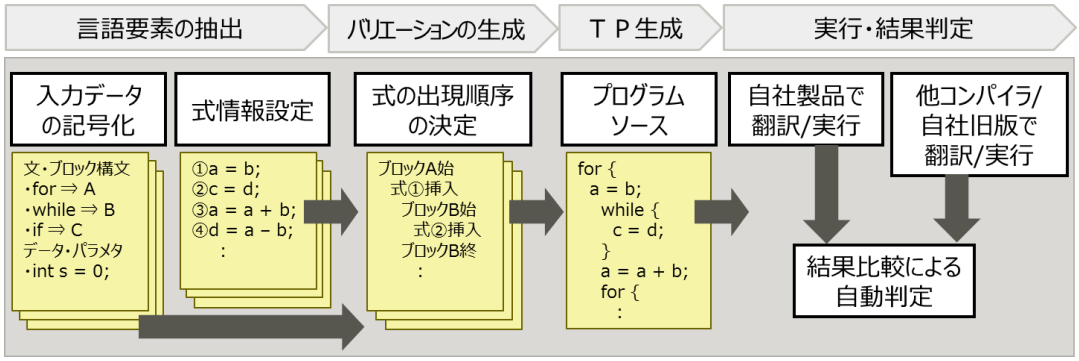

図 89-3 テストプログラム自動生成・実行・判定ツール概要

しかし、活用の実績から以下 2 つの問題が明らかになった。

・問題 1：品質リスクに応じた順番でテストできず、早期の品質確保ができない

・問題 2：NG が大量検出されるが、同一原因か否かの確認は人手であり、工数がかかる

2 つの問題を図 89-4 に示す。問題 1 は、人間がテストする場合、リスクの高いテスト項目から実施し、限られたテスト期間の早い段階でバグを見つけることができるが、自動生成しランダムに流すとテスト期間の終わりに近い段階までバグが見つからず品質確保が遅れるケースがあるということである。問題 2 は、大量のテストを実施するため NG となる項目も大量に検出されるが、それらは同一原因による NG である可能性があり、1 つずつ NG を調査し原因を特定するのに工数がかかるということである。

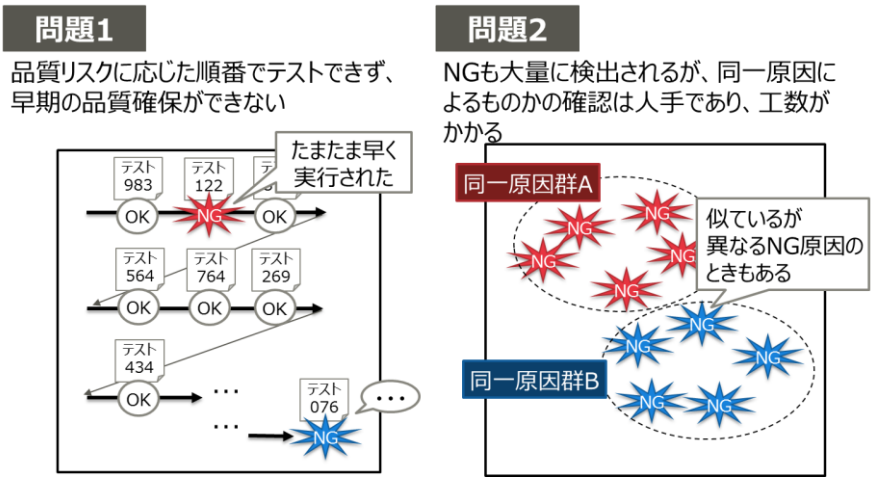

図 89-4 自動生成大量テストにおける 2 つの問題

我々の取り組みの目的は、これらの問題を解決し、図 89-5 に示す自動生成大量テストにおけるありたい姿を実現することである。

・ありたい姿 1：品質リスクの高いテストを先に実行し早期に品質確保

・ありたい姿 2：同一原因 NG の特定を効率化し調査工数を削減

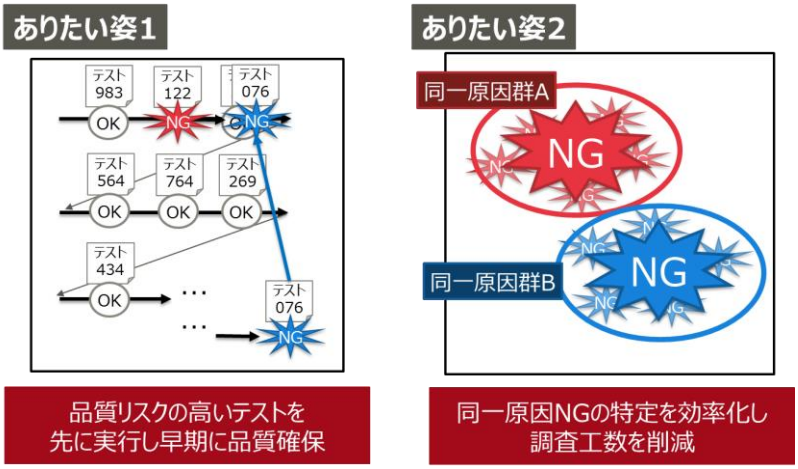

図 89-5 自動生成大量テストにおけるありたい姿

3. 適用手法、効果測定

本章では、2 章で述べた 2 つの問題の解決手法について説明する。

まず、「問題 1 ：品質リスクに応じた順番でテストできず、早期の品質確保ができない」について、以下の取り組み A の適用で解決した。

- 取り組み A：テスト実行順の動的な変更

当社ソフトウェア検証部の過去数十年にわたる評価の経験から、NG となるテスト項目に含まれる要因は、テスト対象ソフトウェアの弱点であることが多い。そこで、それら弱点要因を優先的に組合せテストすることで、テスト実行順が動的に変化し、リスクの高い項目の実行順が早まり、早期のバグ検出が可能になる。手法および効果測定について、3.1 節で詳説する。

次に、「問題 2：NG も大量に検出されるが、同一原因によるものかの確認は人手であり、工数がかかる」について、以下の取り組み B および取り組み C の 2 段階の適用で解決した。

- 取り組み B：同一原因による NG の特定、冗長なテスト項目の排除

同一原因により発生する NG は、テストプログラム内での要因の位置などが少し変化しただけの、類似のテスト項目で起きることが多い。それに着目し、NG となる類似のテスト項目を自動で特定する仕組みを作成し、同一原因で NG となる冗長なテスト項目を特定し実行をスキップすることで、検出 NG の数を減少させ、調査工数を削減する。手法および効果測定について、3.2 節で詳説する。

- 取り組み C：結果比較で NG となるプログラムの最小化

我々のツールでは、テストプログラムが複雑になるよう、大きなプログラムを自動生成しているが、実際に NG に関係する部分は大きなプログラムの一部であり、その箇所だけを抜き出しプログラムを最小化することが可能である。最小化を自動化して、最小化および原因特定の工数を削減する。手法および効果測定について、3.3 節で詳説する。

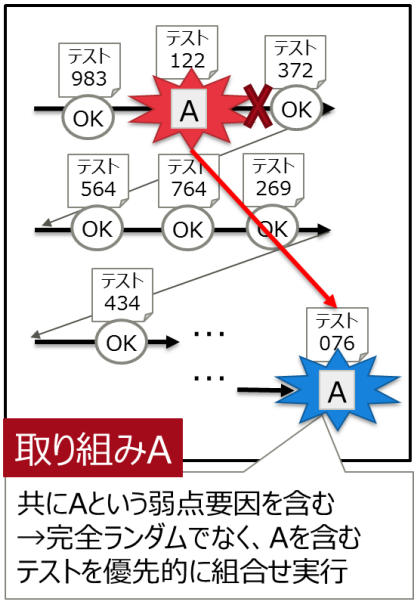

図 89-6 問題 1 解決のための取り組み A

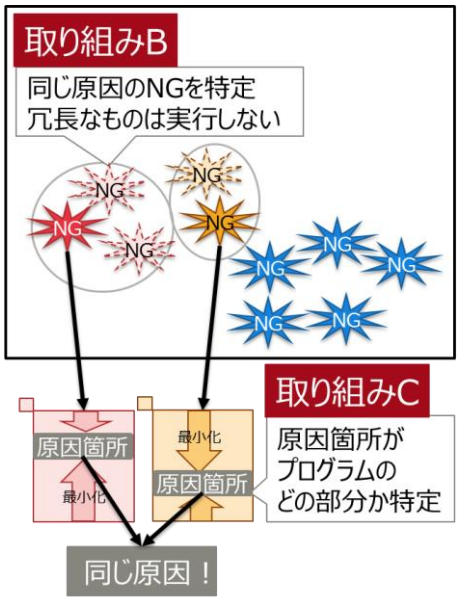

図 89-7 問題 2 解決のための取り組み B、C

3.1. 取り組み A テスト実行順の動的な変更

3.1.1. 手法について

弱点となる要因を優先的に組合せ、品質リスクの高いテストの実行順を早めるため、要因ごとに「優先度」を設定し、この優先度に基づいて要因を組み合わせるアイデアを採用した。

具体的な処理は以下の通りである。処理の番号を対応付けた模式図を図 89-8 に示す。（特許登録済：特許第 5962350 号）

① 開発工程の品質不良部分・フィールド品質分析から、弱点となるテスト要因の点数が高くなるよう、各要因の「優先度」初期値を設定

② NG を検出したら、そのテスト項目に含まれるテスト要因の「優先度」に加点

③ 組合せた要因の「優先度」の合計値が高くなるテスト項目を優先的に実行

④ NG 検出のたびに②で優先度更新、③で組合せ実行を繰返す

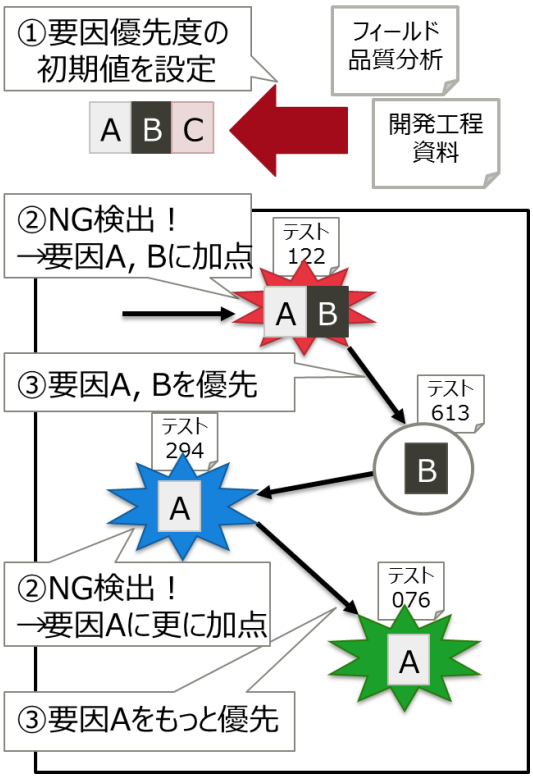

図 89-8 取り組み A テスト実行順の動的な変更の模式図

取り組み A で考慮した点の 1 つは、早く最初の NG を検出できるように、優先度の初期値を設定した点である。「優先度」の初期値をすべてゼロで初期化してから始めると、実行の初期段階で初めの NG が出るまで、優先度が組合せの指標として機能せずランダム組合せと同じ動作になり、なかなか NG を検出できないケースがある。そこで、「優先度」の初期値を、テスト実施前に分かる情報から設定することで、初期段階から優先度が指標として機能し、早く最初の NG を検出できる。

以後、取り組み A での実行方式を「優先実行」と呼称し、従来の完全にランダムに組合せ実行する方式と区別する。

3.1.2. 効果測定

原因判明済みの障害 8 件を含む、旧版の自社コンパイラに対して、優先実行方式と従来の方式でテストを実行する。その際、要因表は 2 つの方式で同一のものを使用する。この条件で、2 つの方式の実行における、原因判明済み障害の検出傾向を比較して、手法の効果を測定し、早期障害検出ができることを確認する。

検出傾向のグラフを図 89-9 に示す。従来方式では 6 件検出まで 230 万本だが、優先実行方式適用時には同じ 6 件検出まで 90 万本であり、約 3 分の 1 の実行本数で障害早期検出ができるという成果が得られた。

一方で、8 件すべての判明済み障害を検出するまでは、従来方式・優先実行方式共に約 400 万本である。図 89-10 に示すように、優先実行方式では、6 件の障害は同じ弱点要因を含むため早期検出できたが、別の要因を含む残り 2 件は実行が後回しになったためである。残り 2 件も含めて 8 件すべてを早期検出するためには、改善が必要である。

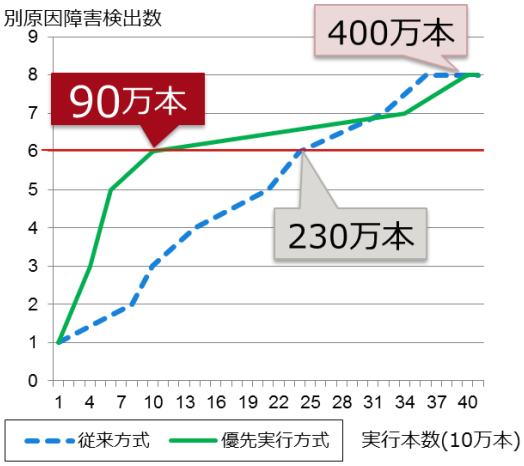

図 89-9 取り組み A 効果測定における障害検出傾向

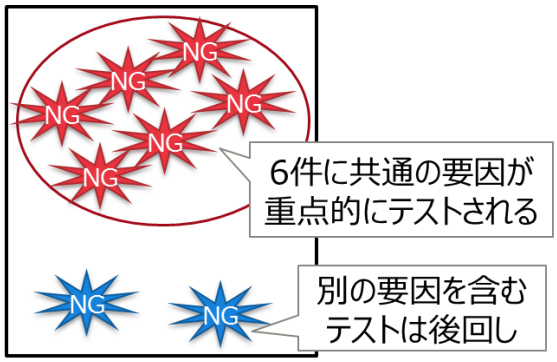

図 89-10 取り組み A の検出イメージ

3.2. 取り組み B 同一原因による NG の特定、冗長なテスト項目の排除

3.2.1. 手法について

テストプログラム内での要因の位置や、変数のデータ型などが少し変化しただけの類似のテスト項目は同一原因であるという考えに基づき、冗長なテスト項目を排除する。NG 検出時に、そのテスト項目の要因を少しずつ変化させてテストを再実行し、同じく NG となるテスト項目を自動的に特定する。

具体的な処理は以下の通りである。図 89-11 に模式図を示す。(特許登録済 : 特許第 5962350 号)

① NG 検出時、要因表からそのテスト項目に含まれる要因の類似パラメータを抽出する。
② 要因のパラメータを変化させてテストを再実行し、NG となるパラメータを記録する。
③ ②で NG となるパラメータでのテスト項目は、①で検出した NG と同一原因と考え、今後、同一パラメータでのテスト項目が生成された場合は、その実施をスキップする。
④ NG 検出のたびに①～③で今後実施をスキップするパラメータの特定を繰返す。

因子	水準1	水準2	水準3	水準4	水準5	水準6
型	char	short	int	long	float	double

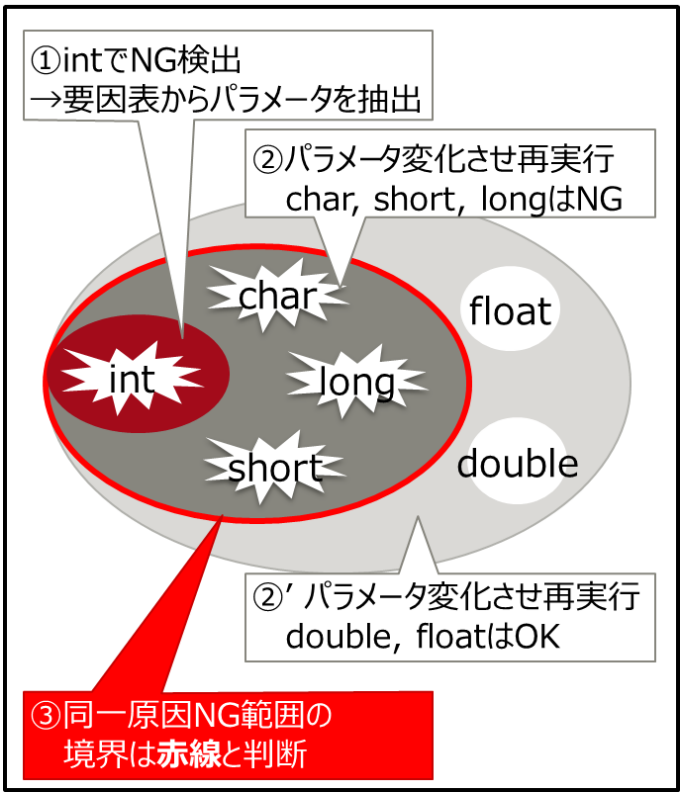

図 89-11 取り組み B 同一原因による NG の特定、冗長なテスト項目の排除の模式図

3.2.2. 効果測定

原因判明済みの障害 4 件を含む、旧版の自社コンパイラに対して、取り組み B を適用した場合と適用しない場合でテストを実行する。その際、要因表は 2 つの方式で同一のものを使用する。この条件で、2 つの方式の実行における、同一原因によるものを含む NG の総検出数を比較して、手法の効果を測定し、冗長なテスト項目が排除され工数削減ができることを確認する。

図 89-12 に 1000 万本走行時の総 NG 検出数の比較を示す。取り組み B 適用前は 5100 件の NG が検出されたのに対し、取り組み B 適用後は 480 件に減少し、適用により NG 検出量を約 10 分の 1 に削減することができた。確認すべき NG の数が 10 分の 1 に減少したため、調査の負担も大きく軽減することができた。

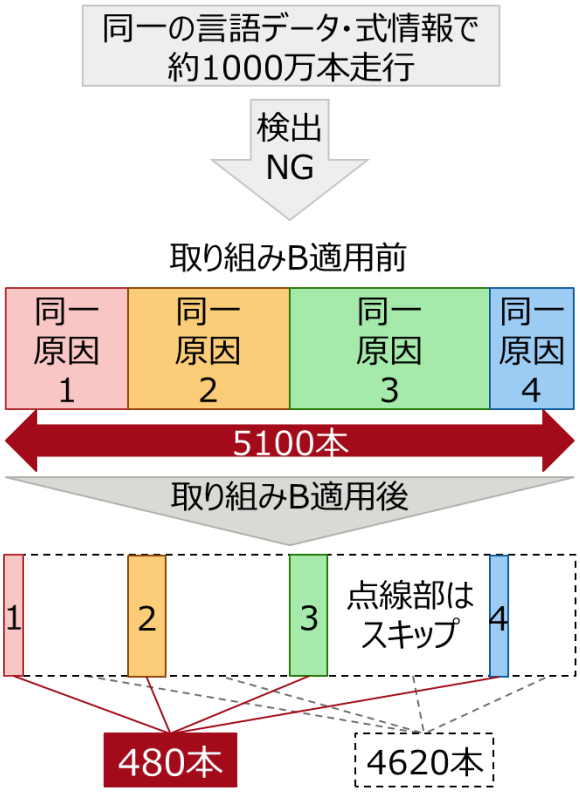

図 89-12 取り組み B 効果測定における総 NG 検出数の比較

この後、これらの検出 NG を取り組み C により最小化して調査しやすくしてから原因特定する。しかし、完全な同一原因特定ができれば、検出 NG の数は 4 件になるはずだが、実際は 480 件までしか削減できず、まだ多くの同一原因 NG が残る。これは取り組み B の同一原因特定技術がまだ不完全であり、改善すべき余地があることを示している。

3.3. 取り組み C 結果比較で NG となるプログラムの最小化

3.3.1. 手法について

テストプログラムの最小化により、NG に関係ない部分を削除し、NG が起きる部分のみからなるプログラムが得られる。図 89-13 に示すように、同一原因で NG となるプログラムであれば類似の形に変形されるため、大きなプログラムのまま調査する場合に比べ、調査の負担を軽減できる。

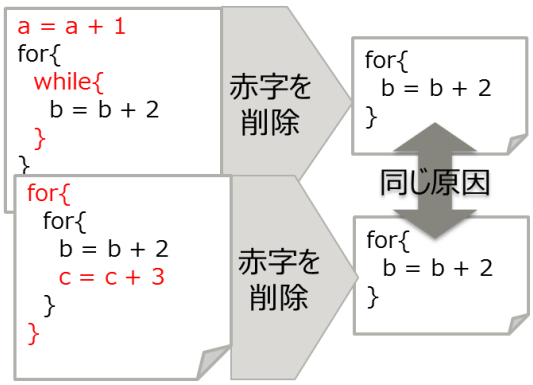

図 89-13 テストプログラム最小化について

そのため従来は、最小化を人手で実施してから同一原因かの確認をしていた。最小化を自動化する既存の手法としては Delta-debugging[1]がよく知られている。これは、図 89-14 に示すように、プログラムを小さくする変換(1 行削除など）を加えて何度もテストを再実行しその都度 NG が再現するかをチェックすることで、NG に関係する部分を特定し、NG が発生する最小の形を特定する手法である。この手法を我々の自動生成ツールが生成したテストプログラムに適用し、自動最小化ができないか考えた。

しかし、実際の自動生成プログラムは、図 89-15 に示すように変数定義・ループ構造などを含み、単純な 1 行ずつ削除という変換では、変数定義を削除してしまい未定義参照が起きたり、ループの始点のみを削除してループのネスト関係の対応が取れなくなったりして、コンパイルエラーとなるケースが多く、適用が難しい。構文解析を実施すればこれらを解決するための情報は得られるが、これはコンパイラフロントエンドの機能であるため、テスト対象であるコンパイラとは別の信頼できる品質のコンパイラを準備する、または自ら構文解析機を構築する必要があり、コストが高い。

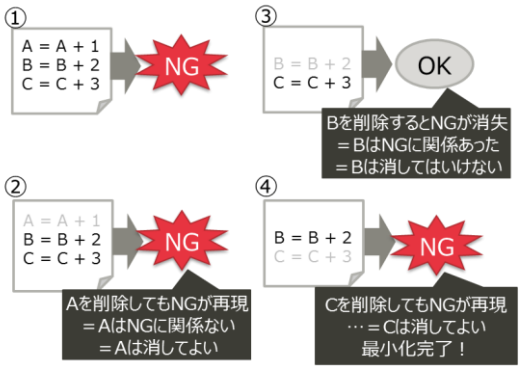

図 89-14 Delta-debugging[1]について

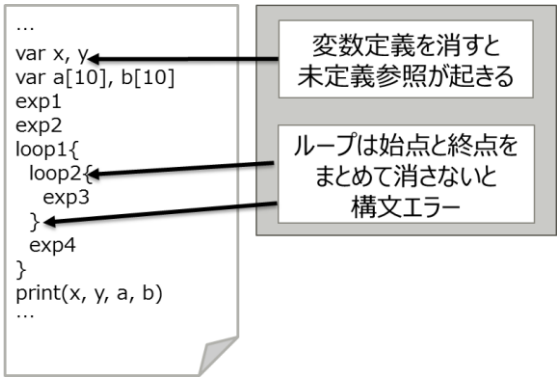

図 89-15 テストプログラム最小化時の留意点

そこで我々は、構文解析を行わず最小化を実施する手法を提案した。我々のテスト項目自動生成ツールの概要図におけるプログラム生成までの部分を図 89-16 に示す。我々のツールは、入力データ・式情報から実プログラムを生成するときに、まず行ごとに種類(定義・式・ループ始点・ループ終点・表示のいずれか)という抽象的な情報に基づき組合せを実施し、次に種類に合致する具体的なプログラム、図 89-16 例 1 のようにループ始点ならば ”for {“ など、図 89-16 例 2 のように式ならば ”a = a + b” などを当てはめていく。そのため、各行がどの種類かという抽象的な中間情報を保持している。この中間情報を出力し再利用することで、構文解析を行わず最小化の実施を可能とした。

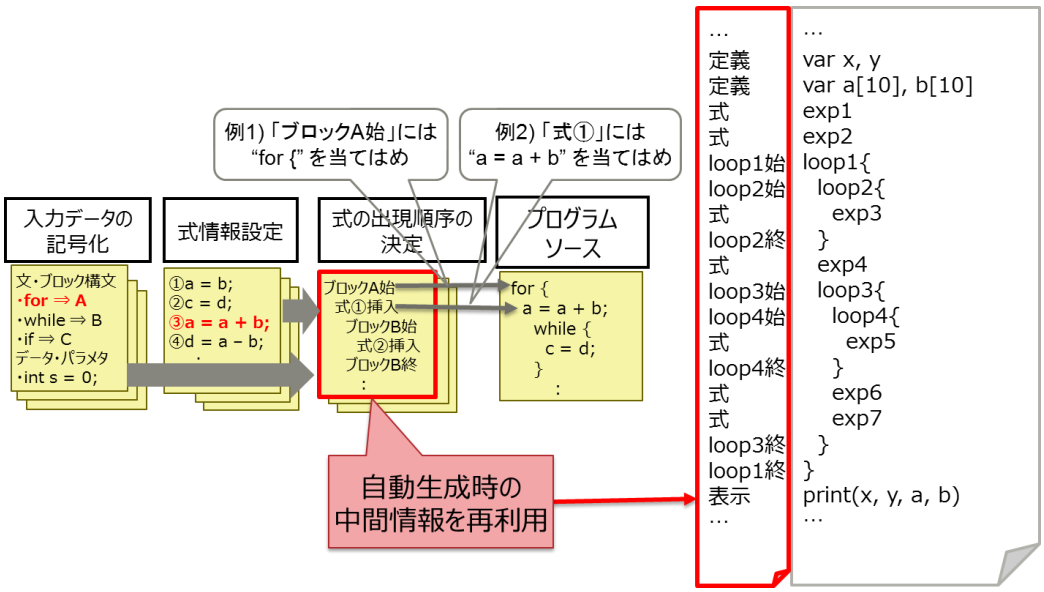

図 89-16 自動生成ツールにおける中間情報を利用した式の当てはめ概要

具体的な処理は以下の通りである。模式図を図 89-17 に示す。

① テストプログラム 1 行に対応する種類情報を参照し、分岐。

a. 種類＝定義または種類＝表示のとき：

何もしない。

b. 種類＝式のとき：

その行を削除しテストを再実行する。結果が OK の場合、削除した行をもとに戻す。NG の場合は削除したままにする。

c. 種類＝ループ始点のとき：

その行および対応するループ終点行をまとめて削除しテストを再実行する。

結果が OK の場合、削除した行をもとに戻す。NG の場合は削除したままにする。

② 次の行に対し、①を再実行。

プログラムの終点に到達すれば終了。

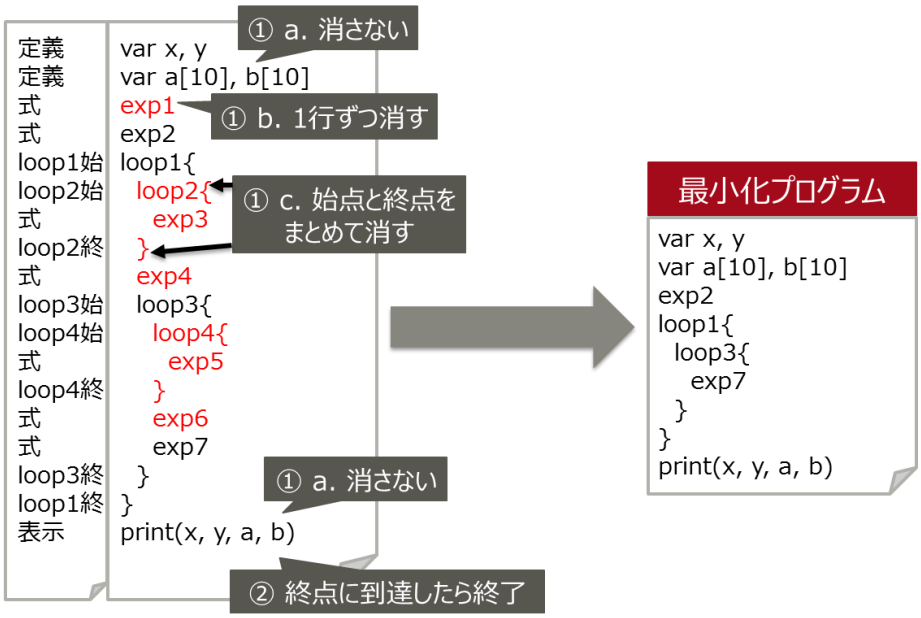

図 89-17 取り組み C 結果比較で NG となるプログラムの最小化の模式図

3.3.2. 効果測定

本手法による、最小化前後のプログラム行数および自動最小化の導入前後の作業工数を比較して、最小化技術および工数削減の効果を測定する。

まず、最小化技術について述べる。自動最小化を取り組み B の出力である 480 本の NG プログラムに対して実施し、平均して最小化前は約 70 行だったプログラムが、最小化後は約 30 行になった。そのうち 5 本のプログラムをピックアップし自動最小化前後のプログラムの行数を比較した結果を図 89-18 に示す。我々のテストプログラム自動生成ツールにおいては、すべてのプログラムに定義・表示の行を固定で 10 行ほど挿入しているため、実際に可変となるランダム生成されるプログラム部分は約 60 行から約 20 行と、3 分の 1 に削減することができた。

最小化前（行）	最小化後（行）
71	32
75	29
69	29
77	34
69	31

図 89-18 最小化によるプログラム行数の変化

次に、工数削減について述べる。以前は最小化を人手で実施していた。これは、ループ構造

の対応を取りながらプログラムの一部を削除し、コンパイル実行して NG が再現するか消失するかを確認する、細かな判断を何度も繰り返す作業で、ミスが発生しやすい。編集ミスにより NG が再現しなくなると、元の大きなプログラムから最小化をやり直す手戻りが起きるケースもあり、実際には平均して 1 本あたり約 60 分の工数がかかる、負荷の大きな作業だった。

しかし、自動最小化の導入により、翻訳実行が 1 回あたり 1 秒かかり、それを平均して 60 回ほど繰り返すことで最小化したプログラムが得られるため、プログラム 1 本あたり約 1 分で最小化できるようになった。これにより、単純計算で工数が 60 分の 1 に削減された。実際は、NG プログラムは数百本出るため、手作業で 1 本 60 分ずつかけて全て最小化することは現実的には不可能だったが、自動最小化により 480 本の最小化が約 8 時間で完了するため、従来不可能だったことが容易に可能になったといえる。

4. 結論

本章では、3 章で紹介した取り組みにより、2 章で設定した 2 つの問題を解決し、ありたい姿にどこまで近づくことができたかについて概観する。2 つの問題とそれに対する取り組み結果を図 89-19 に示す。

まず、「問題 1 ：品質リスクに応じた順番でテストできず、早期の品質確保ができない」に対しては、取り組み A の効果測定により判明済みの障害 8 件のうち 6 件の 75%の障害を、3 分の 1 の実行本数で検出でき、早期検出に有効であることを実証した。

また、「問題 2 ：NG も大量に検出されるが、同一原因によるものかの確認は人手であり、工数がかかる」に対しては、取り組み B・取り組み C の効果測定により、確認すべき NG の本数は 10 分の 1 になり、プログラム行数は 3 分の 1、縮小にかかる時間は 60 分の 1 に削減できることを実証した。

以上から、問題 1・問題 2 ともに、おおむね解決でき、ありたい姿を実現できたと言える。

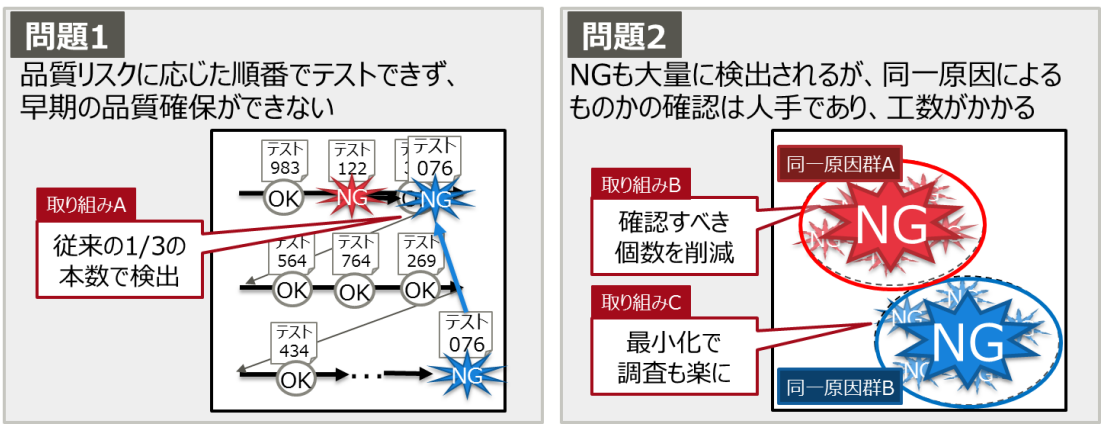

図 89-19 問題とそれに対する取り組み結果

5. 今後の展開

本章では、今後の展開として、5.1 節および 5.2 節で 3 章における取り組み A および取り組み B の残課題解決に向けた展望についてそれぞれ述べ、5.3 節でコンパイラ以外への本手法の展開について述べる。

5.1. 取り組み A の残課題解決

「取り組み：A テスト実行順の動的な変更」における残課題は、優先実行により、判明済み障害 8 件のうち 6 件は従来方式より早く検出できたが、残り 2 件について早期検出できず、従来方式と同等の検出時期となったことである。この理由は 3.1.2 項で述べたとおり、6 件に共通の弱点要因は重点的にテストされたが、残り 2 件はそれと関係ない要因が含まれる項目だったため、残り 2 件の実行順を早めることができなかったためである。(図 89-10 参照)

この課題を解決するために、優先実行方式において 6 件検出後に約 300 万本もの間、新原因 NG が検出されない、検出停滞期間があることに着目する。この停滞期間においては、類似の弱点要因を含む NG は 6 件以外に無いにも関わらず、その弱点要因を優先的に組合せ、既存と同一原因の NG のみを検出している状態となっている。

課題解決のアイデアは、図 89-20 に示すように、既存と同一原因の NG のみが検出され、新原因 NG が検出されない状態を、取り組み B の同一原因特定技術を用いて判定し、一旦優先実行方式を中断して、優先度の高い弱点要因ではなく、別の要因を組合せる実行方式に切り替え、別原因の NG を見付けるというものである。この新しい実行方式および優先実行からこの方式に切り替えるタイミングは現在検討中である。

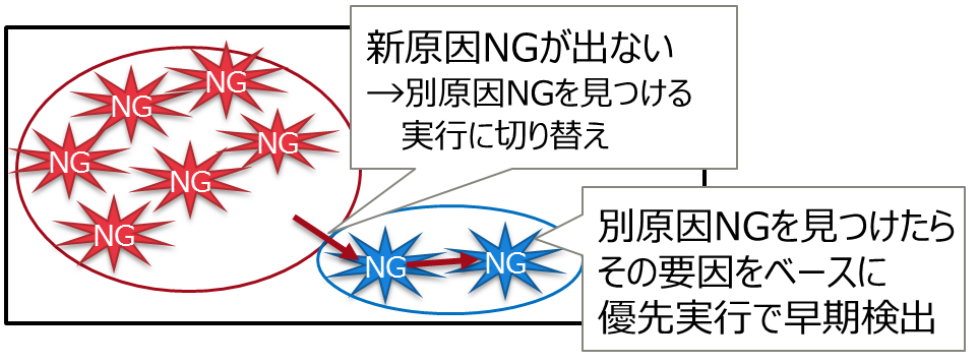

図 89-20 取り組み A 残課題解決のアイデア

5.2. 取り組み B の残課題解決

「取り組み B：同一原因による NG の特定、冗長なテスト項目の排除」の残課題は、判明済み障害が 4 件含まれる旧版コンパイラを対象としたため、完全な同一原因特定ができれば検出 NG の数は 4 件になるが、実際は 480 件までしか削減できず、同一原因特定技術がまだ不完全ということである。

この課題の解決に向け、NG 特定技術が不完全であるという現状をより正確に分析する。特

定が不完全であることによる NG 検出への悪影響は以下 2 つがあると考える。模式図を図 89-21 に示す。

① 範囲が狭く、すべての同一原因 NG を特定しきれず漏れが発生する

② 範囲が広く、別原因 NG も同一原因 NG と判断してスキップしてしまう

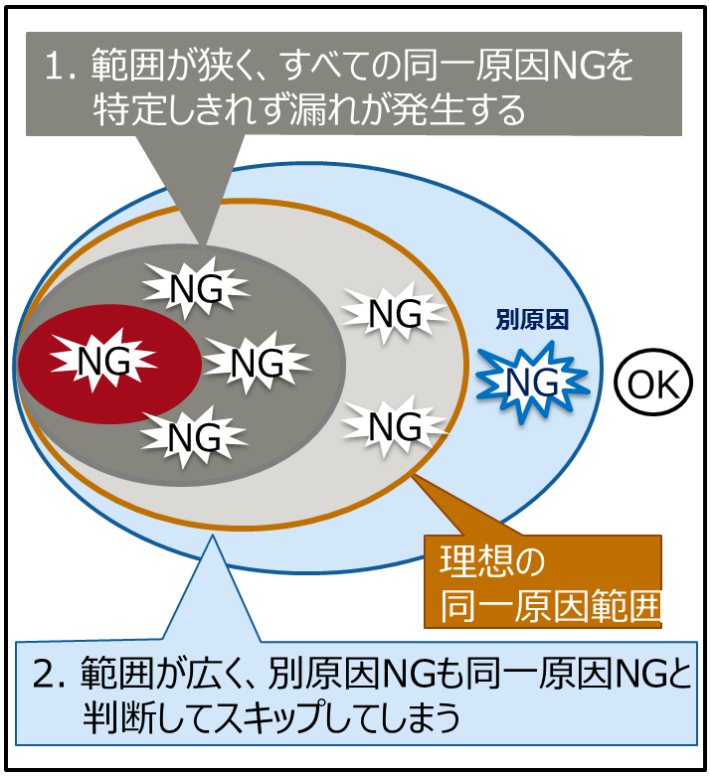

図 89-21 取り組み B 特定技術が不完全である影響について

①は、取り組み B の効果測定による検出傾向から明らかである。取り組み B により、NG 検出時にそれと同一原因 NG と判断する条件を生成するが、その条件でカバーできる範囲が狭く、次に発生した NG が先に発生した NG と同一原因にも関わらず、条件をすり抜けて検出されてしまい、480 件の NG が検出される結果となったためである。

では、同一原因 NG と判断する条件を広げればよいかというとそうではなく、広げすぎると②の現象が起きてしまう。②は、範囲を広く取りすぎて、次に発生した NG は、先に発生した NG と別原因にもかかわらず、同じ原因と誤判断して実行をスキップしてしまう現象である。例えば、取り組み B の効果測定と同等の条件において、同一原因 NG と判断する範囲を広く取ると、検出数は 50 件に減少したが、その中には 4 種類ではなく 3 種類の原因からなる NG しか含まれない、ということが起きるかもしれない。

現状の効果測定では、単純に総 NG 検出数の変化にのみ着目しており、上記の条件すり抜け・誤判断といった観点で評価ができていない。これらの影響をより厳密に測定し、特定技術の洗練について検討中である。

5.3. コンパイラ以外への取り組みの展開

本取り組みは、コンパイラの言語仕様のテストを対象としたが、他のソフトウェアにも展開可能な取り組みと考える。言語仕様や多様なパラメータが存在し、要因組み合わせによる大量のテストの実施が必要な分野には効果的な取り組みである。

例えば、ツールにより自動生成した SQL 文・Web API リクエストのテストにおいては、自動生成による大量テスト技術の展開が可能かつ効果的であると考える。以下にその理由を示す。

まず、SQL 文や、Web API でよく用いられるデータ記述言語である XML や JSON などは、言語仕様を持つ形式的な言語であり、我々の自動生成ツールと同様の考え方で、言語仕様に基づく大量のテスト項目を自動生成できる。

次に、現在、多くのアプリケーションは図 89-22 に示すような構造であり、ユーザが直接、単純な SQL 文や Web API のリクエストを記述することは少ない。フロントエンドでユーザが入力したデータを基にして、内部で機械的に生成した長く複雑な SQL・リクエストでデータのやり取りがされることが多く、データベースサーバや API サーバがそのような複雑なデータを正しく処理できるかという観点でのテストは重要である。

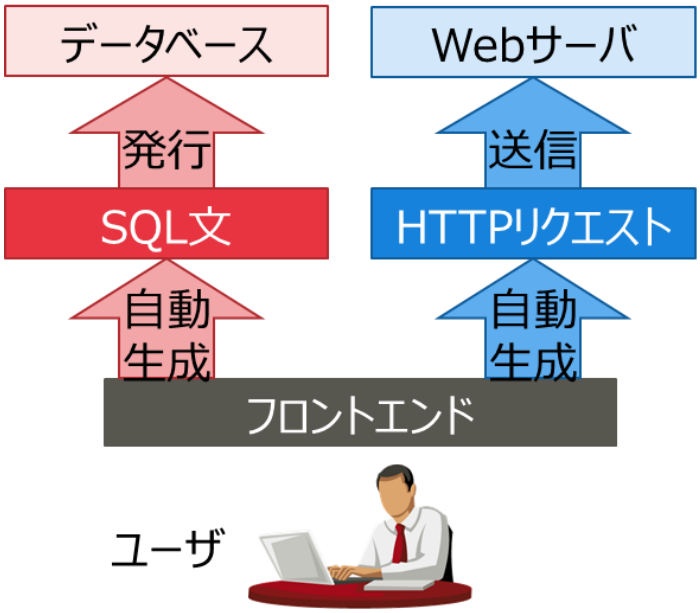

図 89-22 SQL 文・HTTP リクエストを用いる実アプリケーションの例

一例として、Web API において、HTTP リクエストのボディに JSON 形式のデータを持つ場合の、テスト項目自動生成について図 89-23 に示す。

これは、図 89-3 に示したコンパイラのテストプログラム自動生成ツールの枠組みに、Web API テストを当てはめたものである。

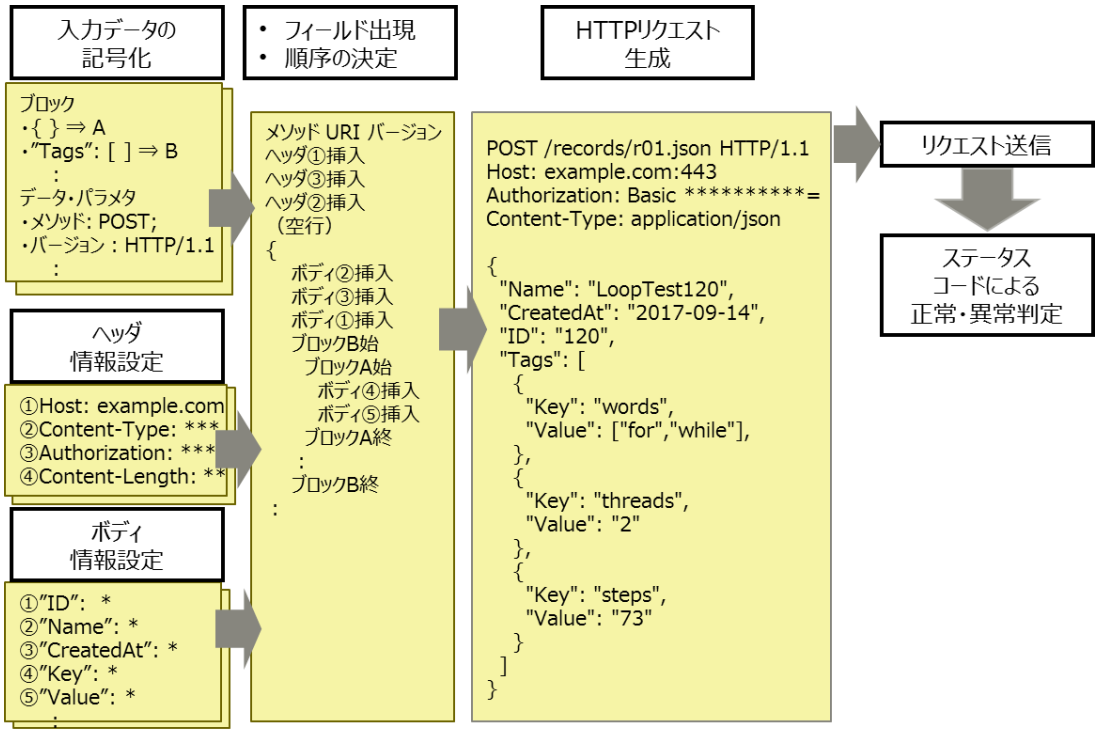

図 89-23 Web API における HTTP リクエスト自動生成テストへの適用例

プログラミング言語も、データ記述言語も、ループやリストに代表される、ネスト可能なブロック構造により言語を記述する点において共通のため、ブロック構造として JSON のペアや配列構造をループ構造の代わりに用いることで、言語仕様に沿った複雑なテストデータを生成できる。これにより、同一の枠組みでテストが可能なため、優先実行による早期品質確保・同一原因 NG の特定・最小化による調査工数削減といった今回の取り組みも展開可能である。これを実証するため、コンパイラ以外のソフトウェアへの手法の展開および効果測定を検討中である。

参考文献

[1] Delta Debugging, https://www.st.cs.uni-saarland.de/dd/

独立行政法人情報処理推進機構 技術本部 ソフトウェア高信頼化センター (IPA/SEC)