C-013

メタモルフィックテスティングの 2 次元物理エンジンへの適用
Applying metamorphic testing to a 2D physics engine

武村 賢弥 † 土屋 達弘 †
Kenya Takemura Tatsuhiro Tsuchiya

1. はじめに

プログラムの出力が正しいかどうかを判定する機構をテストオラクルと呼ぶ。テストオラクルが存在しないか、現実的に実現することが困難な場合、プログラムを non-testable という [1]。また、このようにオラクルを得ることが困難なことをオラクル問題という。例えば、科学計算のプログラムの多くは、未知の結果を得ることを目的としているため、その出力の正しさを直接判定することは通常困難であり、そのため non-testable なプログラムである [1, 2]。本研究では、科学計算の一例でもある数値シミュレーションを取り上げ、そのテストにおいてオラクル問題を解決する方法としてメタモルフィックテストが利用できることを例を通して示す。

メタモルフィックテストとは、複数の異なる入力に対する結果を比較することで、テストオラクルなしでプログラムのテストを行う手法のことである。メタモルフィックテスティングの応用例としては、サーチエンジン [3, 4]、機械学習プログラム [5, 6]、SAT ソルバ [7]、パスプランニング [8, 9] 等のテスティングが挙げられる。

この研究では、数値シミュレーションの例として、科学的応用やゲームなどに幅広く用いられている、物理シミュレーションを対象とする。より具体的には、2 次元物理エンジンに対して、メタモルフィックテストを自動的に実行する枠組みを構築する。さらに、いくつかのメタモルフィックリレーションを提案し、構築した枠組みを用いて、提案したメタモルフィックリレーションに基づいてテストを行う。

2. メタモルフィックテスティング

2.1 テストオラクル

あるプログラムについてテスティングを行うことを考える。そのプログラムの出力が正しいことを確かめるには、あらかじめ入力と真に正しい出力の組を知っておく必要がある。このような正しい入出力の組等を用いて、プログラムの出力結果が正しいかどうかを判定する機構のことを、テストオラクルと呼ぶ。テストオラクルの出力とテスト対象のプログラムの出力を比較することで、テスティングを行うことができる。しかし、多くのプログラムでテストオラクルを得ることは非常に難しく、テストオラクルが存在しない non-testable なプログラムもある。このことをオラクル問題 [10, 11] という。

2.2 メタモルフィックテスティング

メタモルフィックテスティングは擬似オラクルの一種と言えるが、新たにオラクルとなるプログラムを作るのではなく、変更を加えたテストセットに対する出力を利用することで、テスト対象のプログラム自身を擬似オラクルとして用いるという特徴がある。メタモルフィックテスティングでは、入力 A_1、A_2 と得られる出力 B_1、B_2 について、プログラムが正しければ成立する関係を、メタモルフィックリレーションとして定める。メタモルフィックリレーションを用いて、次のようにメタモルフィックテスティングを実行する。

入力 A_1 に対して、メタモルフィックリレーションに基づいて、別の入力 $A_2 = f(A_1)$ に変換する関数 f を定義する。メタモルフィックリレーションには、f が入力 A_1、A_2 の定義域について常に成り立ち、かつ変換が容易となるようなものを選択する。また、このようにして得られた 2 つの入力 A_1，A_2 に対して、テスト対象であるプログラムを実行して得られた出力をそれぞれ B_1 、B_2 としたとき、プログラムが正しければ $B_2 = f'(B_1)$ が成り立つことが期待できる関数 f' を、メタモルフィックリレーションに基づいて設定する。もし、$B_2 \neq f'(B_1)$ であればテスト対象のプログラムに誤りがあったと結論づけられる。

このように、メタモルフィックテスティングでは、出力の正しさが直接判定できなくても、f, f' を適切に定めることができればプログラムの誤りを検出することが可能となる。

3. 提案手法

本研究では、2D 物理エンジンに適用可能なメタモルフィックリレーションを 3 種類設計し、それぞれに基づいたテスト手法を提案する。それぞれの方法と元となるデータによる実験の様子を、図 1 に示した。

3.1 2D 物理エンジン

物理エンジンとは、計算機上で古典力学の計算をシミュレートし、その結果を端末上に表示するソフトウェアである。特に 2 次元座標空間を対象に計算を行うものを 2 次元物理エンジン (2D 物理エンジン) と呼ぶ。本論文が想定する 2D 物理エンジンでは、シミュレーション開始時に、オブジェクトの位置、形状、角度、質量、摩擦係数とともに、場の重力ベクトルが指定できるものとする。シミュレーション開始時のオブジェクトの位置、形状、角度と重力ベクトルがプログラムへの入力であり、シミュレーション中の各時点におけるオブジェクトの位置を出力と考える。

3.2 提案手法 1

1 つめのメタモルフィックリレーションとして、ある点を定め、その点を中心にすべてのオブジェクトと重力ベクトルを回転させた状態を入力としたとき、出力も回転対称となるという性質を考える。元になる入力から得られる出力と、入力を回転させて得られた別の入力から得られる出力とを、後者を逆回転させ前者に一致するかを検査することで、テストを実現する。

†大阪大学 大学院情報科学研究科 Graduate School of Information Science and Technology, Osaka University

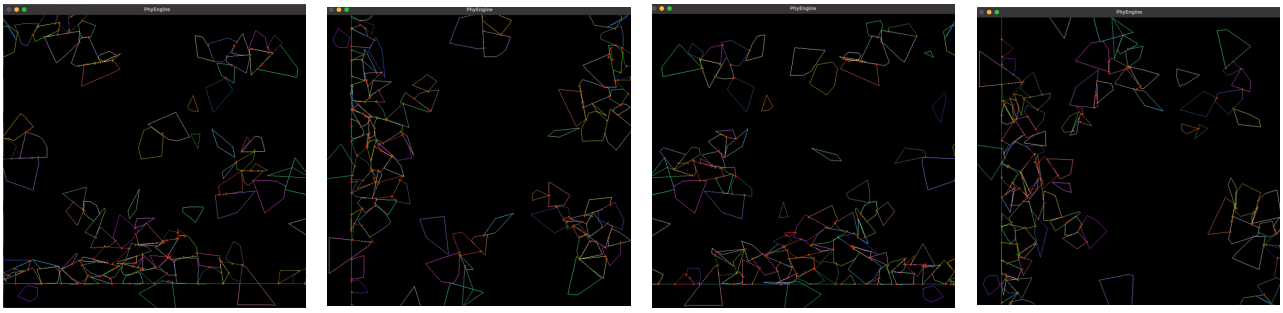

図 1: 左から順番に、MR 適用なし、提案方法１、提案方法２、提案方法３での実験を行ったスクリーンショット。

座標 (x_0, y_0) にある物体を、座標 (x_c, y_c) を中心にして θ 回転させるとき、回転後の物体の座標 (x_1, y_1) と (x_0, y_0) の間には、以下の関係が成り立つ。

$$\begin{pmatrix} x_1 - x_c \\ y_1 - y_c \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x_0 - x_c \\ y_0 - y_c \end{pmatrix} \quad (1)$$

回転させるとき、物体自体の向きも回転させないと正しい結果が出ないことに注意する。θ に -1 をかけることで、(x_1, y_1) から (x_0, y_0) を求めることもできる。以下に逆回転操作を行うことで、元の状態を常に復元できることを示す。

座標 (x_1, y_1) にある物体を、座標 (x_c, y_c) を中心にして $-\theta$ 回転させた座標を、(x_2, y_2) とする。(x_2, y_2) は、メタモルフィックリレーション 1 を用いて式 (2) のように表せる。

式 (2) より $(x_2 - x_c, y_2 - y_c) = (x_0 - x_c, y_0 - y_c)$ が成り立ち、よって $(x_2, y_2) = (x_0, y_0)$ である。上記の議論よりこのメタモルフィックリレーションを用いることで、容易に変換と結果の検証を行うことができることが分かる。

$$\begin{aligned} &\begin{pmatrix} x_2 - x_c \\ y_2 - x_c \end{pmatrix} = \begin{pmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{pmatrix} \begin{pmatrix} x_1 - x_c \\ y_1 - y_c \end{pmatrix} \\ &= \begin{pmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x_0 - x_c \\ y_0 - y_c \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 - x_c \\ y_0 - y_c \end{pmatrix} \\ &= \begin{pmatrix} x_0 - x_c \\ y_0 - y_c \end{pmatrix} \end{aligned} \quad (2)$$

3.3 提案方法２

２つめのメタモルフィックリレーションは、ある直線を定め、その線を中心軸にすべてのオブジェクトを対称移動させた入力に対して、出力も線対称になるという関係である。しかし、ある方向に重力がかかっているとき、シミュレーションは常に重力の影響を受ける。このため、重力の影響に左右されないような移動を考えると、重力と垂直な方向への移動に限られる。以下、y 軸の正の方向に重力がかかっているものとして説明する。

座標 (x_0, y_0) にある物体を、直線 $x = a$ を中心軸にして対称移動させるとき、移動後の物体の座標 (x_1, y_1) と (x_0, y_0) の間には、以下の関係が成り立つ。

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} 2a - x_0 \\ y_0 \end{pmatrix} \quad (3)$$

実際には大きさのある多角形の移動を行うため、多角形の頂点情報も同様にして対称移動させる必要があることに注意する。まったく同じ操作を行うことで、(x_1, y_1) から (x_0, y_0) を求めることもできる。以下に操作を 2 回行うことで、元の状態を常に復元できることを示す。

座標 (x_1, y_1) にある物体を、直線 $x = a$ を中心軸にして対称移動させるとき、移動後の物体の座標を (x_2, y_2) とする。(x_2, y_2) は、式 (3) を用いて以下のように表せる。

$$\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} 2a - x_1 \\ y_1 \end{pmatrix} \quad (4)$$

ここで $y_1 = y_0$ なので $y_2 = y_0$ である。以下では x 座標についてのみ考える。上式の x 座標に関する部分だけ取り出すと、以下のようになる。

$$x_2 = 2a - x_1 \quad (5)$$

また、座標 (x_1, y_1) と (x_0, y_0) の間に成り立つメタモルフィックリレーションのうち、x 座標に関する部分だけ取り出すと、以下のようになる。

$$x_1 = 2a - x_0 \quad (6)$$

式 (5) に式 (6) を代入することで、$x_2 = x_0$ を示せる。

上記の議論により、メタモルフィックリレーションを適用した任意の状態から、元の結果を復元できることが分かる。

3.4 提案方法３

3.2 節と 3.3 節の手法では、ともに入力変換の関数の逆関数を出力変換の関数としているため、これらを合成することができる。つまり、回転対称と線対称を組み合わせた関係を第 3 のメタモルフィックリレーションとして考える。これを用いたテストでは、入力のオブジェクトを、直線 $x = a$ を軸として対称移動させた上で、中心座標 (x_c, y_c) を中心に θ だけ回転することで別の入力を得る。そして、その入力から得られた出力を、$-\theta$ 回転と直線に対して対称移動した上で、もとの入力に対する出力と比較する。

4. 実験

4.1 対象とする物理エンジン

オープンソースの 2D 物理エンジンである Impulse Engine[‡]を対象として実験を行った。Impulse Engine は

[‡]https://github.com/RandyGaul/ImpulseEngine

OpenCV ライブラリを使用し、C++で記述されている。Impulse Engine は円と凸多角形の計算をサポートしており、基本的なパラメータを自由に設定することができる。本実験はすべて凸多角形を対象に行った。

実験では、観察する座標平面上の下部に大きな長方形を水平に設置して地面の代わりとした。地面の上方のランダムな位置に、重力の影響を受けない静止した物体を 20 個設置した。フィールドの上方に、観察する対象である、物理演算の影響を受ける物体を 100 個設置した。

4.2 節での自動化を行うために、ソースコードに編集を加え、入力を外部のファイルから受けとるようにした。また、静止状態でないすべてのオブジェクトの座標を、1 度の計算ごとに標準出力に出力した。本論文ではこの 1 回の計算の時間ステップを、1$Tick$ と呼ぶ。

エンジンの実行は、指定した回数の計算を終えたとき終了するものとした。

4.2 実験方法

実験は、macOS Big Sur version 11.1 上で、Python スクリプトを実行することで行った。Impulse Engine のコンパイルは Apple clang version 12.0.0 を用いた。

実験データの生成には、Python3 の疑似乱数ライブラリを用いた。Impulse Engine で必要なパラメータを疑似乱数を用いて生成し、空白区切りでファイルに出力することで実験データを作成した。

物理エンジンのコンパイル、物理エンジンのソースコードの編集は、サブプロセス機能を使って実現した。ソースコードの編集には sed を使用した。また、4.1 節に記述した標準出力は、サブプロセス機能内で外部ファイルに繋げた。

乱数で生成したテストケースの入力に、メタモルフィックリレーションを適用する操作及び、得られた出力に対して再びメタモルフィックリレーションを適用し、元のテストケースの出力と比較する操作は、Python で行った。

また、テストケースを生成する Python の乱数のシード値を 1 ケース生成するごとに変更し、実験を行う Impulse Engine の乱数のシード値は 0 で固定した。

3.2 節の手法については、計算で誤差を出すことを防ぐため 90°、 180°、 270°、 360° の回転に限定し、さらに回転の中心は画面の中央の座標とした（それぞれの場合を Rotate90, Rotate180, Rotate270, Rotate360 と略記）。

3.3 節の手法については、事前に分かっている Impulse Engine に存在するバグを回避するために、画面の x 座標中央を軸に対称移動させたものに限定した (Mirror と略記)。

3.4 節の手法については、3.2 節と 3.3 節のものに準じた。すなわち、画面の x 座標中央を軸に対称移動させたものを、90°、180°、270° 回転させたものに限定した（Mirror+Rotate90, Mirror+Rotate180, Mirror+Rotate270, Mirror+Rotate360 と略記）。

4.3 結果

図 2 に、メタモルフィックテスティングの結果を示す。図 2 では、最終状態、すなわち $Tick = 499$ におけるオブジェクトのうち、変換を適用しない元の出力と変換した入力に対する出力を逆変換した結果とを比較した際に、最もユークリッド距離が離れていたものに注目し、シミュレーション時間に伴うこの距離の変化を図示している。

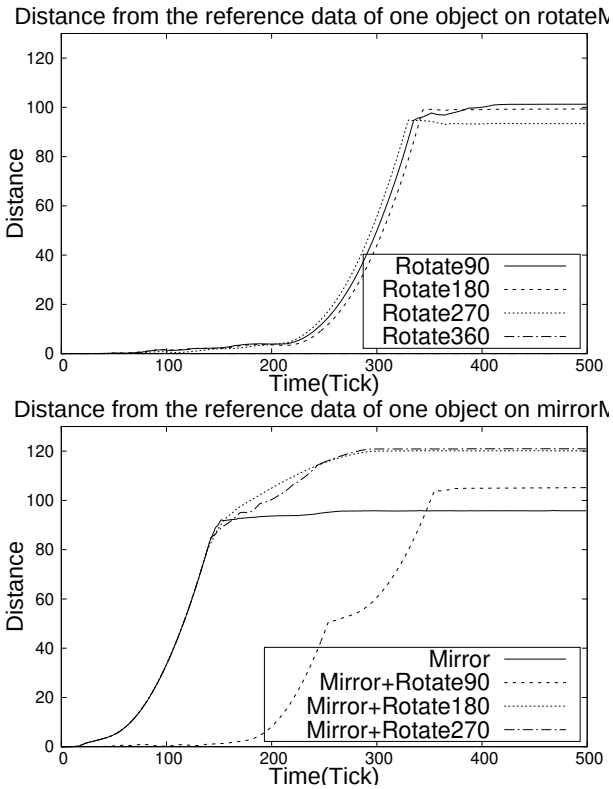

図 2: メタモルフィックテスティングの結果

図 2 上図が 3.2 節の手法による操作を行った群の結果、下図が 3.3 節および 3.4 節の手法による操作を行った群の結果である。縦軸でシミュレーション中の座標の単位でどれだけ距離が離れていたかを表し、横軸でシミュレーションにおける時間変化を表した。3 節によると、理論上誤差が出ないはずなので、非常に大きな誤差が意図せず生じていることが分かる。実際 360° 回転したテストデータに生じた誤差は、10^{-14} ほどで無視できるほど小さい。360° 回転したテストデータを除くそれぞれのケースで、大きく誤差が増えている局面があるが、これは変換を適用していない場合とは異なったタイミングで、別のオブジェクトとの衝突が起きた結果、静止状態にあった物体が本来ある場所から落下したからだと考えられる。実際、大きく増加している部分は 2 次関数で近似できる。

3.2 節の手法に比べて、3.3 節の手法の結果は、各回転テストケース間でのグラフ出力が酷似している。そのため、3.3 節の手法適用時に問題がなかったかどうかを調べるために、追加で検証を行った。結果を図 3 に示す。

図 3 は、3.3 節のメタモルフィックリレーションを適用していないもの (図 2 下図における Rotate90、Rotate180、Rotate270、およびメタモルフィックリレーション適用前のデータ) と、3.3 節のメタモルフィックリレーションを適用したもの (図 2 上図における Mirror、Mirror+Rotate90、Mirror+Rotate180、Mirror+Rotate270) に分類した上で、それぞれの群の各

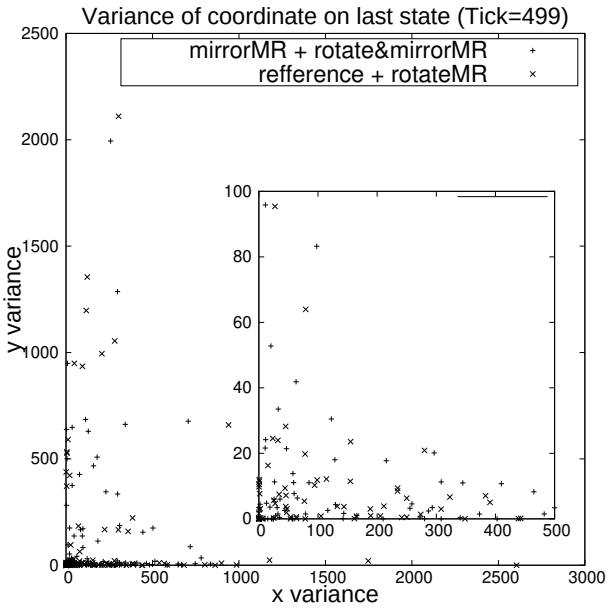

図 3: 対称移動を行った組と行っていない組の座標の分散の比較

オブジェクトについて、最終状態における座標の分散を求めたものである。分散は次元ごとに求め、縦軸に y 座標の分散、横軸に x 座標の分散を取りプロットした。図 3 では、3.3 節の変換の有無によって明確な差は確認できず、したがって図 2 での一致は、偶然同じ同じオブジェクトが同じタイミングで落下したことによって生じたと考えられる。

また、大きな誤差が生じたオブジェクトの数は、最初は 1 つか 2 つであり、時間経過とともに増えて、最終的にはすべてのオブジェクトで無視できない誤差が生じる。

このような大きな誤りが生じた原因は複数考えられるが、蓋然性が高いものは、「ある 1 つのオブジェクトの計算で誤りが生じ、それが全体に伝播した」もしくは「すべてのオブジェクトの計算で細かい誤差が蓄積し、全体に大きな誤差を生んだ」のどちらかである。

いずれにしても、浮動小数点数の計算誤差である可能性が高いと考えられる。

5. おわりに

本研究では、物理エンジンへメタモルフィックテスティングを応用する方法について検討した。その結果、浮動小数点などの細かい計算誤差が存在するかを、ある程度効率的に判別できることが分かった。この結果から、計算の厳密さを高く要求する学術用の物理エンジンなどに対して、誤差の有無を調べるためにメタモルフィックテスティングを適用することは、有用であるといえる。

今回は対称性のある物体配置に対するシミュレーションの振る舞いにのみ注目したので、対称性の下で物体が等しい状態になる性質に基づくメタモルフィックリレーションしか検討できなかった。今後の課題としては、ある種の物理法則を利用したメタモルフィックリレーションを提案し、テストをより多角的に行うことが挙げられる。

参考文献

[1] E.J. Weyuker, “On testing non-testable programs,” Comput. J., vol.25, no.4, pp.465–470, 1982. https://doi.org/10.1093/comjnl/25.4.465

[2] U. Kanewala and J.M. Bieman, “Testing scientific software: A systematic literature review,” Information and Software Technology, vol.56, no.10, pp.1219–1232, 2014. http://www.sciencedirect.com/science/article/pii/S0950584914001232

[3] Z. Zhou, S. Zhang, M. Hagenbuchner, T.H. Tse, F. Kuo, and T.Y. Chen, “Automated functional testing of online search services,” Softw. Test. Verification Reliab., vol.22, no.4, pp.221–243, 2012. https://doi.org/10.1002/stvr.437

[4] Z. Zhou, S. Xiang, and T.Y. Chen, “Metamorphic testing for software quality assessment: A study of search engines,” IEEE Trans. Software Eng., vol.42, no.3, pp.264–284, 2016. https://doi.org/10.1109/TSE.2015.2478001

[5] A. Dwarakanath, M. Ahuja, S. Sikand, R.M. Rao, R.P.J.C. Bose, N. Dubash, and S. Podder, “Identifying implementation bugs in machine learning based image classifiers using metamorphic testing,” Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018, eds. by F. Tip and E. Bodden, pp.118–128, ACM, 2018. https://doi.org/10.1145/3213846.3213858

[6] X. Xie, J.W.K. Ho, C. Murphy, G.E. Kaiser, B. Xu, and T.Y. Chen, “Testing and validating machine learning classifiers by metamorphic testing,” J. Syst. Softw., vol.84, no.4, pp.544–558, 2011. https://doi.org/10.1016/j.jss.2010.11.920

[7] S. Segura, A. Durán, A.B. Sánchez, D.L. Berre, E. Lonca, and A.R. Cortés, “Automated metamorphic testing of variability analysis tools,” Softw. Test. Verification Reliab., vol.25, no.2, pp.138–163, 2015. https://doi.org/10.1002/stvr.1566

[8] M. Lindvall, A. Porter, G. Magnusson, and C. Schulze, “Metamorphic model-based testing of autonomous systems,” 2nd IEEE/ACM International Workshop on Metamorphic Testing, MET@ICSE 2017, Buenos Aires, Argentina, May 22, 2017, pp.35–41, IEEE Computer Society, 2017. https://doi.org/10.1109/MET.2017.6

[9] J. Zhang, Z. Zheng, B. Yin, K. Qiu, and Y. Liu, “Testing graph searching based path planning algorithms by metamorphic testing,” 24th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2019, Kyoto, Japan, December 1-3, 2019, pp.158–167, IEEE, 2019. https://doi.org/10.1109/PRDC47002.2019.00046

[10] S. Segura, “A survey on metamorphic testing,” IEEE Transactions on Software Engineering, vol.42, no.9, pp.805–824, 2016.

[11] E.T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” IEEE Transactions on Software Engineering, vol.41, no.5, pp.507–525, 2015.