Amazon Web Services はどのように形式手法を利用したか

“How Amazon Web Services Uses Formal Methods"

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff

https://dl.acm.org/doi/10.1145/2699417

はじめに

2011 年以来、アマゾン・ウェブ・サービス（AWS）のエンジニア達は、クリティカルなシステムでの設計上の難しい問題を解決するのを助けるために、形式手法とモデル・チェッキングを利用してきた。この論文では、我々のこうした取り組みでの動機と経験、さらには、これらの問題領域でうまくいったこととうまくいかなかったことについて述べる。 著者達の個人としての取り組みを紹介するときには、著者のイニシャルで示している。

AWS では、顧客が簡単に使用できるサービスの構築に努めている。 AWS のサービスの外から見たシンプルさは、実は、複雑な分散システムの隠れた基盤の上に構築されている。この複雑な内部構造は、費用対効果の高いインフラ上でサービスを実行して高可用性を実現しながら、とどまることのないビジネスの成長に対応するために要求されたものである。この成長の例として、AWS が 2006 年に発表した、シンプル・ストレージ・サービス S3 がある。S3 は、発表後の 6 年間で 1 兆個のオブジェクトを格納するように成長した[3]。その後は、1 年も経たないうちに、2 兆個のオブジェクトを格納するように成長し、通常毎秒 110 万のリクエストを処理していた。

S3 は、顧客から託されたデータを保存・処理する多くの AWS サービスの 1 つにすぎない。顧客のデータを安全に保護する上で、AWS のそれぞれのサービスの中核部分は、フォールト・トレラントな分散アルゴリズムに依存している。それらはつぎのようなものである。データの複製、データの一貫性の保持、並列実行制御、自動スケーリング、ロード・バランシング、また、それらの調整タスクがある。研究論文上にはそのようなアルゴリズムが多数あるのだが、それらのアルゴリズムを、まとまりのある一つのシステムに組み合わせるのは、挑戦的な課題である。なぜなら、それらを実世界のシステム上で正しく相互作用させるためには、通常はアルゴリズムを変更する必要があるからである。さらに、我々のシステムは、我々独自のアルゴリズムを必要とすることもわかってきていた。 我々は不必要な複雑さを避けるために一生懸命取り組んでいたのだが、この仕事の本質的な複雑さは依然として高いままである。

システムの複雑さは、設計・コード・操作での人的エラーの確率を高める。システムの中核部分にエラーがあると、データの損失や破損が発生したり、顧客が依拠している他のインターフェイス規約を侵害する可能性がある。そえゆえ、サービスを開始する前に、システムの中核部分が正しいという非常に高い確信を得る必要がある。我々は、業界の標準的な検証手法は必要であるが、十分ではないことに気づいていた。 我々は、深いデザイン・レビューやコード・レビュー、静的コード分析、ストレス・テスト、およ

びフォールト・インジェクション・テストを日常的に使用していたが、複雑な並行フォールト・トレラント・システムでは、微妙なバグが隠れている可能性がある。我々が行った一つの推論は、毎秒数百万のリクエストのスケールで動作するシステムで、「非常にまれ」と思われるイベントの組み合わせの、本当の確率を評価するには、人間の直感はとても貧困であるということであった。

NASA の C. Michael Holloway 氏は、「おおざっぱに振り返って最初にわかることは、事故というものは、ほとんど常に、1 つ以上の起こりうることの誤った評価の結果であると言えることだ。」と語っている。人間が誤りやすいということは、より微妙で危険なバグのいくつかは、設計上のエラーであることを意味する。すなわち、我々はコードで意図した設計を忠実に実装しようとするが、その設計は特定の「まれな」シナリオを正しく処理できないのだ。コードの到達可能な状態の数は天文学的なものであるため、コードのテストは設計における微妙なエラーを見つける方法としては不十分であることがわかった。 そこで、我々は、より良いアプローチを探すこととなった。

本論文の重要な洞察

- 形式手法は、我々が知っている他のどんな技術を通じても見つけることができないデザインを行う。
- 形式手法は、主流のソフトウェア開発で驚くほど容易に実現可能であり、その投資に対するリターンも高いものになる。
- アマゾンでは、形式手法が、パブリック・クラウドサービスを含む複雑な現実のソフトウェアの設計に日常的に利用されている。

正確な設計

システム設計の微妙なバグを見つけるには、その設計の正確な記述が必要である。正確な設計を書くことには、少なくとも 2 つの主要な利点がある。プログラマはより明確に考えることを余儀なくされ、それは、「起こりうる手振れ」を排除するのに役立つ。また、設計が書かれている最中でも、設計エラーをチェックするツールを適用できる。対照的に、従来の設計ドキュメントは、散文的であり、静的な図とおそらくはテスト不可能なアドホック言語の疑似コードで構成されている。そのような記述は正確にはほど遠いものである。多くの場合、それらは、あいまいであるか、規定されるべき重要な側面を欠いている（部分的な障害や並行性の粒度など）

問題のスペクトルのもう一方の端では、最終的な実行可能コードには曖昧さはないが、圧倒的な量の詳細が含まれている。我々は、数百行の正確な記述で設計の本質を捉えることができなければならなかった。我々の設計が複雑なのは避けられないことなので、実装コードのレベルをはるかに超えて、正確なセマンティクスを備えた非常に表現力豊かな言語が必要だった。その表現力は、実世界での計算の並列性とシステムのフォールト・トレランス性をカバーしなくてはならない。その上、我々はサービスを迅速に構築したいので、少数者だけに理解される難解な概念を避け、簡単に学習して適用できる言語が必要だった。ま

た、我々は、ツールの既存のエコシステムがそのまま使えることを強く望んでいた。我々は、投資効率の高い既製の方法を探していた。

我々は、探していたものを TLA+の中に見出した。TLA+は、シンプルな離散数学、すなわち、すべてのエンジニアがよく知っている初等的な集合論と述語論理に基づいた形式的仕様記述言語である。TLA+での仕様は、システムのルールに基づいた可能なすべての振る舞いの集合、すなわちシステムの実行トレースの集合を記述する。我々は、システムに望まれる正しさという諸特性（「何が？」）と、システムの設計（「どのように？」）の双方を記述するのに、同じ言語が使用されていることが有用であることを見出した。TLA+では、正しいという諸特徴とシステム設計は抽象化のはしごの一歩である。TLA+では、正しいという諸特徴は上位レベルを、システム設計とアルゴリズムは中間レベルを、実行可能コードとハードウェアは下位レベルを占めることになる。TLA+は、システムの設計が、システムに望まれる正しさの諸特徴を正しく実装していることをできるだけ簡単に示そうとする。それは、従来から用いられていた数学的推論を通じてか、あるいはまた、TLA+仕様を使って、実行可能なすべての実行トレースにわたって、望ましい正しさの諸特徴を徹底的にチェックする TLC モデルチェッカー[9]のようなツールを通じて行われる。抽象化のはしごは、設計者が現実世界のシステムの複雑さを管理するのにも役立つ。設計者は、システムをいくつかの「中間」レベルの抽象化レベルで記述して、下位レベルごとに異なる目的（細粒度の並行実行の様々な結果や、通信経路のより詳細な振る舞いを理解するなど）を与えることできる。その上で、設計者は、各レベルが上位レベルに関して正しいことを検証できる。抽象化のレベルを自由に選択して調整できることで、TLA+は非常に柔軟である。

最初のうちは、TLA+の構文とイディオムは、プログラマーにはどちらかといえばなじみがないものであった。幸い、TLA+には、C スタイルのプログラミング言語によく似た PlusCal と呼ばれる 2 番目の言語が付属している。それは、式や値に TLA+を使用しているため、C よりは表現力に優れている。PlusCal は、疑似コードを直接置き換えることを目的としている。Amazon のエンジニアには、TLA+を使用するよりも PlusCal を使用する方が生産性が高いことに気づいたものもいた。しかし、他の場合では、TLA+をそのまま使う方が、それのもつ追加の柔軟性が利用できて非常に有用だった。PlusCal はキーを 1 回押すだけで自動的に TLA+に変換されるため、多くの設計では、どちらを選ぶかは好みの問題である。PlusCal ユーザーは、豊富な表現を書くために TLA +に精通している必要がある。これは、TLA +の翻訳を読んでコードの正確なセマンティクスを理解することが役立つ場合が多いためである。さらに、ツール（TLC モデル・チェッカーなど）は TLA +レベルで機能するからである。

実世界のシステムにとっての形式手法

業界には、形式手法は、比較的単純なコードのごく一部を検証するために膨大なトレーニングと労力を必要とするという評判があった。そのため、形式手法に対する投資のリターンは、安全がクリティカルに重要な領域（医療システムや航空電子機器など）でのみ正当化されると考えられていた。 TLA+に関す

る我々の経験は、この認識が間違っていることを示している。この論文の執筆時点(2015 年)では、Amazon のエンジニアは、10 の大規模で複雑な現実世界のシステムで TLA+を使用している。それぞれの取り組みで、TLA+は、重要な価値を追加した。そこでは、TLA+は、他の方法では発見できないのは確かと言える微妙なバグを見つけ、あるいは、正確さを犠牲にすることなく積極的なパフォーマンス最適化を行うのに十分な理解と自信をプログラマに与えた。現在、Amazon には TLA+を使用する 7 つのチームがあり、上級の管理者と技術リーダーの励ましを受けている。エントリー・レベルからプリンシパルまでのエンジニアが、TLA+をゼロから学習し、2〜3 週間で有用な結果を得ることができた。週末や夕方の個人的な時間を使って、支援やトレーニングなしでも、有用な結果をだしたケースもある。

この論文では、形式仕様のソースの断片を引用することをしていない。それらの馴染みのない構文が、潜在的な新規ユーザーに不快感を与える可能性があるためである。潜在的な新規ユーザーにとっては、いきなり形式手法のチュートリアルやサンプルに取り組む前に、この業界での形式手法の価値について聞くことのほうが、メリットがあることがわかっている。読者は、次のものを参照されたい、チュートリアルとしては Lamport らの[11]、この号の 38 ページの Lamport の視点、Amazon のいくつかのより大きな仕様（次の表を参照）とサイズと複雑さが似ている業界の TLA+仕様の例としては Lamport[12]。

Applying TLA+ to some of Amazon's more complex systems.

System	Components	Line Count (Excluding Comments)	Benefit
S3	Fault-tolerant, low-level network algorithm	804 PlusCal	Found two bugs, then others in proposed optimizations
	Background redistribution of data	645 PlusCal	Found one bug, then another in the first proposed fix
DynamoDB	Replication and group-membership system	939 TLA+	Found three bugs requiring traces of up to 35 steps
EBS	Volume management	102 PlusCal	Found three bugs
Internal distributed lock manager	Lock-free data structure	223 PlusCal	Improved confidence though failed to find a liveness bug, as liveness not checked
	Fault-tolerant replication-and-reconfiguration algorithm	318 TLA+	Found one bug and verified an aggressive optimization

我々は、TLA+が我々の問題領域では効果的であることを見出したのだが、他にも多くの形式仕様言語とツールがある。その一部については後で説明する

付随的なメリット

TLA+は、システム設計のより良い方法への移行を支援してきた。エンジニアは自然と、システムの「ハッピーケース」、すなわち、エラーが発生しない処理の設計に集中する。こうした幸せなケースが圧倒的に最も一般的なケースであるため、このことは理解できる。そのコードパスは、顧客の問題を解決し、適切に実行し、リソースを効率的に使用し、ビジネスに合わせて拡張する必要がある。これらのすべてのチャレンジは大事なことだ。ハッピーケースの設計が完了すると、エンジニアは、個人的な経験と同僚やレビューアーの経験に基づいて、「何が問題になる可能性があるか」を考える。この時、エンジニアはこれらの可能性のある問題のシナリオに、直感を優先して、おそらくはそれらが起きる確率を統計的に考えて、問題の緩和策を追加する。ただ、ほとんどの場合、想像できないほど多くのシナリオがあるため、エンジニアはイベントの「非常にまれな」組み合わせを処理できていないところで停止する。

それとは対照的に、形式仕様を使用する場合には、「正しく進むには何が必要か」を正確に述べることから始める。我々は、最初に、次の 2 種類の正しさの特性を定義することで、システムが何をすべきかを指定する。

安全性。システムにそれを行うことが許されていること。たとえば、常に、コミットされたすべてのデータが存在し正しいものであること。または同じことだが、いかなる時でも、コミットされたデータがシステムによって失われたり破損したりすることはないということ。

生存性。 システムが最終的には行わなければならないこと。たとえば、システムがリクエストを受け取るたびに、最終的には、そのリクエストに応答する必要があるということ。

こうした正しさの特性を定義した後で、我々は設計の抽象的なバージョンとその動作環境の抽象的なバージョンを正確に記述する。 システムが依存する環境のすべての特性を明示的に指定することによって、「正しく進むべきこと」を表現する。

このような特性の例としては、「通信チャネルに障害が発生していない場合、メッセージはそれに沿って伝播される」、「プロセスが再起動されていない場合、意図的な変更をのぞいて、そのローカル状態を保持する」などである。 次に、設計が環境内のすべての動的イベントを正しく処理することを確認することを目的として、ネットワークエラーと修復、ディスクエラー、プロセスのクラッシュと再起動、データセンターの障害と修復、人間のオペレーターによる行動など、起こりうる各イベントの影響を特定する。次に、モデルチェッカーを使用して、動作環境でのイベントの組み合わせやインターリーブに関係なく、その環境でのシステムの仕様が、選択された正しさの特性を実装していることを確認する。こうした厳密な「正しく進むため何が必要か」というアプローチは、アドホックな「うまくいかないものは何か」というアプローチよりもエラーが発生しにくいことがわかる。

さらなるメリット

また、形式仕様を書くことは、システムの存続期間にわたって利益をもたらすことがわかる。 アマゾンのすべての製品サービスは、数年前にリリースされたものも含め、常に開発が続けられている。 顧客が要求した新しい機能を追加し、規模の大幅な増加に対応できるようにコンポーネントを再設計し、ボトルネックを取り除くことでパフォーマンスを向上させる。 これらの変更の多くは複雑であり、ダウンタイムなしで実行中のシステムに加える必要がある。 我々の最優先事項は常に製品版のシステムでのバグの発生を回避することであるため、「この変更は安全であるか？」という問いに答えなければならない。コアシステムの正確でテスト可能なモデルを使用することの主な利点は、たとえ深い変更であっても、それが安全であることをすばやく確認できることである。あるいは、その変更が実際にシステムに害を及ぼすことなく、安全でないことを学習できることである。いくつかのケースでは、微妙だが深刻なバグが本番環境に到達するのを防いだ。 他のケースでは、革新的なパフォーマンスの最適化（ロックの削除やその範囲を狭めたり、メッセージの順序付けの制約の緩和など）を行うことができた。それらの変更のモデル・チェックなしでは、我々はそうした最適化を敢えて行わなかっただろう。システムの正確でテスト可能な記述は、スプレッドシートが財務モデルの what-if ツールであるのと同様に、設計の what-if ツールになっている。 このようなツールを使用してシステムの動作を調査すると、システムの設計者の理解を向上できることがわかった。

さらに、設計の正確でテスト可能な、十分にコメントされた記述は、優れた形式のドキュメントである。というのも、AWS システムのライフタイムには期限がないため、このことは、重要である。時間が経過し、ビジネスの成長とともにチームも成長する。そのため、我々は定期的に新しい人々をシステムに慣れさせる必要がある。この教育は効果的でなければならない。 微妙なバグを作らないようにするには、すべてのエンジニアがシステムの同じメンタルモデルを持つことが必要なのだが、その共有モデルは、精密で正確で完全である必要がある。 エンジニアはさまざまな方法でメンタルモデルを形成する。互いに話し合ったり、設計ドキュメントを読んだり、コードを読んだり、バグ修正や小さな機能を実装したりする。 しかし、会話や設計のドキュメントはあいまいであったり不完全であったりする。実行可能コードが大きすぎてすぐに理解できなかったり、あるいはそのコードは意図したデザインを正確に反映していない場合があるのだ。 対照的に、形式仕様は、正確で短く、ツールを使用して調査や実験ができる。

形式仕様が向いていないこと

我々は、大規模な分散システムに関する 2 つの主要なクラスの問題に関心を持っている。第一のものはプログラムのバグとオペレーターのエラーである。それらは、システムの論理的な意図から逸脱する原因となる。第二のクラスは、複雑なシステムの驚くべき「持続的なパフォーマンスの突然の低下」をもたらすものだ。それは フィードバック・ループを必然的に含んでいる。我々は、形式仕様を使用して、第一の

クラスの問題を見つける方法を知っている。 ただし、第二のクラスの問題では、論理的なバグが含まれていなくても、システムが機能しなくなる可能性がある。一般的な例は、サーバーでの一時的なスローダウン（おそらく Java ガベージコレクションが原因）によってクライアントでタイムアウトが発生し、クライアントがリクエストを再試行してサーバーに負荷を追加し、さらにスローダウンする場合である。そのようなシナリオでは、システムは結局のところ動いていて、 論理的なデッドロック、ライブロック、またはその他のサイクルでスタックしているわけではない。しかし、顧客の観点からみれば、許容できない応答時間が続くため、事実上利用できないことになる。 TLA+を使用して、安全性の特性として応答時間の上限を指定できるのだが。しかしながら、AWS システムは、ハード・リアルタイムスケジューリングやその保証をサポートしないインフラストラクチャ（ディスク、オペレーティングシステム、ネットワーク）上に構築されているため、システムに、リアルタイムの安全性を指定することは現実的ではなかった。 我々は、非常に短い期間の応答の遅さはエラーとは見なさないソフト・リアルタイムシステムを構築している。 ただし、深刻なスローダウンが長引けば、エラーと見なされる。 我々は、ツールがそのような緊急の行動を予測することを可能にする実際のシステムをモデル化する実行可能な方法をまだ知らない。 他の手法を使用して、これらのリスクを軽減する。

形式手法への第一歩

後から考えると、Amazon の形式手法への道のりは、次のように一直線のように見える。エンジニアリングの問題があり解決策を見つけた。 ただ、現実は多少異なったものだった。 この取り組みは、著者の一人 Chris Newcombe（ 以下 C.N ）.が設計およびレビューしたいくつかの分散システムの品質に対する不満と、それらのシステムの構築に使用された開発プロセスとツールへの不満から始まった。 システムは成功したと見なされたが、バグと運用上の問題は持続した。問題を軽減するために、このシステムでは、実績のある方法、本番環境で有効になっている広範な契約アサーションを使用してバグの症状を検出し、メカニズム（「回復指向コンピューティング」[20]など）を使用して、バグがトリガーされたときの影響を最小限に抑えようとしていた。 ただ、リアクティブ・メカニズムでは、顧客データに永続的な損傷を与えるバグのクラスから回復することはできなかった。 代わりに、そのようなバグが作成されるのを防ぐ必要があった。

バグを防ぐためのテクニックを探すとき、C.N は、当初は形式手法を検討していなかった。というのも、形式手法は、小さな問題にのみ適し、投資収益率が非常に低いという、見方が一般に広がっていたからである。形式手法に対する偏見を克服するには、実際のシステムで機能する証拠が必要だった。 この証拠は、Zave によって与えられた [22]。彼は、Alloy と呼ばれる言語を使用して、Chord と呼ばれる分散システムのメンバーシップ・プロトコルの重大なバグを見つけたのだ。Chord は MIT の専門家グループによって設計され、SIGCOMM 2011 カンファレンスで「10 年間のテスト」賞を受賞し、業界のいくつかのシステムに影響を与えて成功していた。Zave の成功は C.N.が、Alloy の評価を行うことを動機づけ

た。彼は、比較的大規模な自明ではない並行アルゴリズム[18]の Alloy 仕様を書き、そのモデルチェックを行うことで Alloy の評価を行なった。我々は、Alloy 言語の多くの特徴を好ましいと思っている。集合と関係で構成される抽象的なシステム状態の「実行トレース」の強調もその一つだ。ただし、AWS の多くのユースケースでは、我々は、Alloy の表現力が十分でないこともわかった。 たとえば、Alloy では、豊かなデータ構造（複数のフィールドを持つ入れ子のレコードを含む動的シーケンスなど）を表現する実用的な方法を見つけることができなかった。

Alloy の限られた表現力は、Alloy Analyzer ツールによる分析の特定のアプローチの結果であると思われる。 この制限は、Alloy の概念モデル（システム状態の「実行トレース」）が原因ではないように見える。 この仮説は C.N. が、Alloy と同様の概念モデルを持ち、システム状態を記述するためのより豊富な構成を備えた言語を探すことを動機づけた。C.N. は、最終的には、これらの性質を持つ言語に出くわした。彼は、我々の問題領域の標準的アルゴリズムである Paxos コンセンサス・アルゴリズムに関する Lamport の論文[12]のアペンディックスに TLA+仕様を見つけたのである。

TLA+がこのように広く使用されているアルゴリズムの設計者によって作成されたという事実は、TLA+が実際のシステムで機能するという確信を我々に与えた。 DEC/Compaq のエンジニアのチームが TLA+を使用して、マルチコア CPU の Alpha シリーズの複雑なキャッシュ・コヒーレンシープロトコルの仕様と検証をしている[5,16]ことを知ったとき、我々はより自信を持った。仕様の 1 つ[13]を読んだところ、豊富なメッセージパッシング、きめ細かい並行実行性、複雑な正確性プロパティを含む高度な分散アルゴリズムが記述されていた。残ったのは、TLA+が実際の障害モードを扱うことできるかどうかという問題だけだった。 （Alpha キャッシュ・コヒーレンシー・アルゴリズムは障害を考慮していない。）
Lamport の Fast Paxos 論文[12]から、TLA+は高レベルの抽象化でフォールト・トレランスをモデル化できることを知り、TLA+が低レベルの障害をモデル化できることを示す他の論文[15]を見つけたとき、我々は TLA+が使えると、さらに確信した。

C.N. は、Alloy で書いたのと同じ自明でない並行アルゴリズムの仕様を書いて TLA+を評価した[18]。Alloy と TLA+はどちらも問題を処理できたが、比較すると、TLA+は Alloy よりはるかに表現力があることが明らかになった。 この違いは実践的には重要である。 TLA+で作成した実際の仕様のいくつかは、Alloy では実現不可能だった。 我々は、当初は、TLA+について反対の考えを持っていた。というのも、TLA+は非常に表現力があるので、この言語で表現できるすべてのものをモデル・チェッカーが評価することには望みがないと考えていた。 しかし、これまでのところ、我々は、常に我々の意図を表現する方法を見つけることができた。それは、明確で直接的でモデルチェックが可能な方法だった。

Alloy と TLA+を評価した後、C.N は、アマゾンの同僚たちに TLA+を採用するよう説得しようとした。しかし、エンジニアには必要に迫られない限り、そのようなことをする暇な時間はほとんどなかった。幸いなことに、ある必要が生じようとしていた。

Amazon での最初の大きな成功

2012 年 1 月、Amazon は DynamoDB を立ち上げた。これは、強力なデータ一貫性を約束しながら、複数のデータセンター間で顧客データを複製する、スケーラブルで高性能な「No SQL」のデータストアである[2]。 この要件の組み合わせは、大規模で複雑なシステムにつながるものであった。 DynamoDB のレプリケーションとフォールト・トレランスのメカニズムは、著者の一人 Tim Rath (以下 T.R.)が作り上げたものだ。 T.R.は、製品版のコードの正しさを検証するために、シミュレートされたネットワーク・レイヤーを使用して広範なフォールト・インジェクションテストを実行し、メッセージの損失、複製、並べ替えをコントロールした。 システムはまた、多くの異なるワークロードの下で、実際のハードウェア上で長期間ストレス・テストされた。そのようなテストは絶対に必要であるが、我々は、設計の微妙な欠陥を明らかにできない場合があることを知っていた。 DynamoDB の設計を検証するために、T.R は、システムの正しさの詳細な非形式的な証明を書いた。これによって、設計の初期バージョンにいくつかのバグを実際に見つけることができた。しかし、我々は、従来の非形式的証明では非常に微妙な問題を見逃す可能性があることも学習していた[14]。設計で最高レベルの信頼性を達成するために、T.R は、TLA+を選択した。

T.R.は TLA+を学び、これらのコンポーネントの詳細な仕様を 2 週間ほどで書いた。仕様をモデルチェックするために、10 個の cc1.4xlarge EC2 インスタンスのクラスターで実行される TLC モデルチェッカーの分散バージョンを使用した。それぞれのインスタンスは、8 つのコアとハイパースレッドで、23GB の RAM が搭載されていた。モデルチェッカーは、アルゴリズムの小さな複雑な部分がシステムの十分に大きなインスタンスに対して期待どおりに機能することを検証し、システムが正しいことの高い信頼性を提供した。T.R.は、次に、より広範なフォールト・トレラント・アルゴリズムをチェックした。その時、モデルチェッカーは、特定の障害シーケンスと回復手順が他の処理とインターリーブされると、データが失われる可能性があるバグを発見した。これは非常に微妙なバグだった。 このバグを示す最も短いエラートレースでも、35 個の高レベルのステップが含まれていた。 そのような複合したイベントがありそうもないからといって、それはそのようなバグに対する防御にはならないのだ。 歴史的に、AWS エンジニアは、このバグをトリガーする可能性のあるイベントと同じくらい複雑なイベントの多くの組み合わせを観察してきた。このバグは、広範なデザインレビュー、コードレビュー、テストを気づかれることなく通過していた。T.R.は、こうした従来の領域でのやりかたでは、我々は、より多くの作業を行ってもそれを見つけることはできなかっただろうと確信している。 モデルチェッカーはその後、他のアルゴリズムに深刻で微妙な 2 つのバグを発見した。 T.R. はこれらのすべてのバグを修正した。モデルチェッカーは最終的なアルゴリズムを非常に高い信頼性で検証したのだ。

T.R. は、DynamoDB の作業を開始する前に TLA+について知っていたなら、最初からそれを使用していたと言う。 彼は、正式な TLA+仕様の作成と確認に費やした投資は、彼が非形式的な証明を作成して確認する作業よりも信頼性が高く、時間もかからなかったと考えている。 従来のような非形式的証明を

書く代わりに TLA+を使用することで、この例のように、システムの正しさに対する信頼が高まるだけでなく、市場投入までの時間が短縮される可能性があるのだ。

DynamoDB のローンチ後、T.R は、データセンター間でデータを移行できるようにする新機能に取り組んだ。 彼はすでに既存の複製アルゴリズムの仕様を持っていたので、T.R は、この新しい機能を仕様にすばやく組み込むことができた。モデルチェッカーは、初期の設計で微妙なバグが発生する可能性があることを発見したが、修正は簡単であり、モデルチェッカーは結果のアルゴリズムを必要なレベルの信頼性で検証した。 T.R. は引き続き TLA+とモデルチェックを使用して、最適化と新機能の両方に対する設計の変更を検証した。

多くのエンジニアを説得する

DynamoDB での成功は、Amazon のより広範なエンジニアリング・コミュニティに TLA+を提示するのに十分な証拠を与えてくれた。 これは、形式手法の目的と利点をいかにしてソフトウェアエンジニアに伝えるかというチャレンジングな課題を引き起こした。 エンジニアは「検証」ではなくデバッグの観点から考えるため、プレゼンテーションでは「設計のデバッグ」と呼んだ[18]。こうしたメタファーを続けながら、エンジニアは、「徹底的にテスト可能な疑似コード」と呼ぶと、TLA+の概念と実用的な価値をより簡単に把握できることに気づいた。 形式手法は実践的ではないと広く見られていたため、最初は「形式」、「検証」、「証明」という言葉を使うのを避けた。また、我々は、最初に TLA が何を表すのかについて言及することは避けた。

我々のプレゼンテーションを見た直後に、S3 に取り組んでいるチームが、TLA+を使用して新しいフォールト・トレラント・ネットワーク・アルゴリズムを検証するための支援を求めてきた。 アルゴリズムのドキュメントは、多くの大きく複雑な状態機械図で構成されていた。 状態マシンをチェックするために、チームは可能な実行を総当たり的に調査する Java プログラムを書くことを検討していた：それは、本質的には、モデルチェックのハードワイヤード形式である。代わりに TLA+を使用することで、彼らはその労力を回避することができた。著者の一人 Fan Zhang（以下 F.Z.）は、二週間ほどで 2 つのバージョンの仕様を作成した。 この特定の問題については、彼女は PlusCal の方が TLA+より生産性が高いと考えている。我々も、エンジニアは、PlusCal を使い始める方が簡単であることが多いことを確認している。 モデルチェックにより、アルゴリズムに 2 つの微妙なバグが明らかになり、F.Z が両方の修正を確認することになった。 F.Z.は、 次に、仕様を使用して設計を実験し、新しい機能と最適化を追加した。モデルチェッカーはすぐに、これらの変更の一部にバグが発生することを明らかにした。

この成功から AWS の管理層は、S3 に取り組んでいる他のチームに TLA+を推奨することとなる。 S3 のチームのエンジニアは、2 つの追加の重要なアルゴリズムと 1 つの新しい機能の仕様を作成した。 F.Z. は、彼らが彼らの最初の仕様を書く方法を教えるのを助けた。 自分自身が TLA+にはまだ新しいエンジニアが、他のエンジニアに TLA+を教えることができることに気づいたことは我々には励みになった。こ

れは、Amazon と同じような大規模な組織では、この手法の採用を迅速に拡大するためには、重要なことであった。

著者の一人 Bogdan Munteanu (以下 B.M.)は、 そのようなエンジニアの一人だった。 彼の最初の仕様は、微妙なバグを含むことが知られているアルゴリズムに関するものだった。 このバグは気付かれずに複数のデザイン・レビューとコードレビューを通過し、数か月のテストを経て初めて表面化した。B.M.は 2 週間かけて TLA+を学び、仕様を作成した。それを使用して、TLC モデルチェッカーは数秒でバグを発見した。チームはすでにバグの修正をおえ、レビューしていたため、B.M は、提案された修正を含むように仕様を変更した。モデルチェッカーは、別の実行トレースでまだ問題が発生していることを検出した。より強力な修正が提案され、モデルチェッカーは 2 番目の修正を検証した。B.M.は後に別のアルゴリズムの別の仕様を書いた。 その仕様ではバグを見つからなかったが、この仕様が解決しようとしたアルゴリズムのドキュメントに、いくつかの重要な曖昧さが含まれていることが明らからかになった。

ある程度独立しているが、TLA+に関する内部プレゼンテーションを見た後、著者の Marc Brooker と Michael Deardeuff (以下 M.B と M.D.)は、自分たちで PlusCal と TLA+を学び、車内のさらなる説得や支援なしにそれぞれのプロジェクトでそれらを使用し始めた。 M.B. は、PlusCal を使用して 3 つのバグを見つけ、彼は、TLA+での彼の個人的な実験について、Amazon の外で公開ブログを書いた。[7]
M.D.は PlusCal を使用してロックフリーの並行アルゴリズムをチェックし、次に TLA+を使用して AWS の最も重要な新しい分散アルゴリズムの 1 つに重大なバグを発見した。 M.D.はバグを修正も、その修正を検証した。 独立して、C.N は、M.D.が作成した仕様とはスタイルがまったく異なる同じアルゴリズムの仕様を作成したが、どちらもアルゴリズムに同じバグが見つかった。 これは、TLA+を使用する利点がエンジニア間のばらつきに対して非常に堅牢であることを示唆している。 両方の仕様は、アルゴリズムの重要な最適化によってバグが発生しなかったことを検証するために後で使用された。

Amazon のエンジニアは引き続き TLA+を使用している。そこでは、従来の散文的な設計ドキュメントを最初に作成してから、その一部を段階的に PlusCal または TLA+に改良するというやりかたを採用している。 この方法では、たとえ完全な仕様やモデルのチェックを行わなくても、設計に関する重要な洞察が得られることがよくある。 あるケースでは、C.N は、別の Amazon エンジニアが設計したフォールト・トレラント・レプリケーション・システムの散文的な設計を改良した。
C.N.は、 2 つのレベルで並行性の仕様を記述しモデルチェックした。 これらの仕様は、彼が設計を深く理解することを助けた。彼は、システムの書き込み待ち時間を大幅に削減する主要なプロトコル最適化を提案することができた。我々はまた、TLA+はデータモデリングに優れたツールであることも発見した。それは、リレーショナルデータベースや「No SQL」データベースに対してスキーマを設計するのと同じように。我々は、TLA+を使用して、標準の多重度制約や外部キー制約よりもはるかに豊富なデータのセマンティック不変量を備えた重要なスキーマを設計した。我々は、次に、データの主要な操作のいくつかの高レベルの仕様を追加した。それは、スキーマの修正と調整に役立った。この結果は、データモデルを

システム全体の抽象化の別のレベルと見なすことができることを示唆している。 また、TLA+は、設計者がシステムのスケーラビリティを向上させるのに役立つ可能性があることも示唆している。 スケーラビリティのボトルネックを解消するために、設計者はしばしば、アトミック・トランザクションを非同期ワークフローを通じてチェーン化されたより細かい操作に分割する。 TLA+は、分離と一貫性に関してこのような変更の結果を調査するのに役立つ。

よくある質問

TLA+について学習する際、エンジニアは、よく次のように聞く。「実行可能コードが検証済み設計を正しく実装していることをどのようにして知るのか？」 その答えはわからない。 それにもかかわらず、形式手法は依然として複数の点で役立つ。

設計を正しく行う： 形式手法は、エンジニアが設計を正しく行うのに役立つ。それは、コードを正しくするために必要な最初のステップである。 設計が壊れている場合、コードはほぼ確実に壊れている。コーディングの間違いが設計の間違いを補うことはほとんどありそうもない。さらに悪いことに、エンジニアはコードが「正しい」と信じ込んでいる可能性がある。なぜなら、（壊れた）設計を正しく実装しているように見えるからである。エンジニアは、コーディングにフォーカスしている間、設計が正しくないことに気付くことはほとんどなかった。

より良い理解をうる： 形式手法は、エンジニアが設計をよりよく理解するのに役立つ。 設計の理解を改善することだけが、エンジニアが正しいコードを得るチャンスを増大させる。

より良いコードを書く： 形式的手法は、エンジニアがアサーションの形でより適切な「自己診断コード」を作成するのに役立つ。 独立した証拠[10]と我々自身の経験は、アサーションの普及がコードのエラーを減らすための良い方法であることを示唆している。 アサーションは、システム全体の不変の小さなローカルな部分をチェックする。 優れたシステム不変条件は、システムが機能する根本的な理由を捉えている。 システムが不変の状態を継続的に維持する限り、システムは安全特性に違反する可能性のある問題を発生させない。課題は、安全特性に違反しないことを保証するのに十分強力な、優れたシステム不変性を見つけることである。 形式手法は、エンジニアが強力な不変性を見つけるのに役立つ。したがって、形式手法は、アサーションを改善してコードの品質を向上させるのに役立つ。

我々は、実行可能コードが高レベルの仕様を正しく実装していることを検証したいと思うし、あるいは、仕様からコードを生成することさえしたいと思うのだが、Amazon で構築されているものと同じくらい大規模で複雑な分散システムを処理できるツールはなかった。 我々は日常的に従来の静的分析ツールを使用しているが、それらは主にコード内の「ローカル」問題の検出に限定されており、高レベルの仕様への準拠を検証することはできない。

TLC モデルチェッカーを使用して、コードをテストする設計の「エッジ・ケース」を見つける研究[21]があった。これは、有望と思われるアプローチである。 ただし、Tasiran らの論文[.21]は、ハードウェア設計をカバーしていたが、我々は、この方法をソフトウェアに適用することをまだ試みていない。

TLA+の代替

多くの形式仕様の手法がある。 我々はそのいくつかを評価し、調査結果を公開した Newcombe[19]。そこでは、形式手法がこの業界で成功するために重要であると考える要件を示している。 TLA+がこれらの要件を満たしていることがわかったので、我々は形式手法の評価を中止した。我々の目標は、常に実践的なエンジニアリングで、網羅的な調査ではないからである。

関連する仕事

業界で複雑な分散システムの設計を検証するために高レベルの形式仕様を使用することについては、ほとんど公開された文献は無いようにみえる。Farsite プロジェクト[6]は複雑であるが、我々がこの論文で記述したシステムのタイプとは異なる。それは、明らかに商業的には公開されていない。 Abrial[1]は、商用の安全が重要な制御システムでのアプリケーションを引用しているが、それらは我々の問題領域ほど複雑ではないようである。 Lu et al[17]は、フォールト・トレラントな分散ハッシュテーブルのよく知られたアルゴリズムの事後検証を記述し、Zave[22]は、別の同様のアルゴリズムを記述しているが、これらのアルゴリズムが商用製品で使用されているかどうかはわからない。

結論

形式手法は、AWS で大きな成功を収めており、他の手法では発見できなかった、微妙で重大なバグが本番環境に到達するのを防ぐのに役立っている。 これらは、品質を犠牲にすることなく、複雑なアルゴリズムを積極的に最適化するのに役立った。 この論文の執筆時点では、7 つの Amazon チームが TLA+を使用しており、そのすべてチームがそうすることに価値を見出しており、より多くの Amazon チームがそれを使用し始めている。 TLA+を使用することで、システムの市場投入時間と品質の両方が向上するであろう。 経営陣は、新機能やその他の重要な設計変更に関する TLA+仕様を作成することをチームに積極的に奨励している。 年次計画では、マネージャーはエンジニアリングの時間を TLA+に割り当てている。

我々の結果は有望であるが、いくつかの重要な警告が残っている。 形式手法は、システム自体ではなくシステムのモデルを扱うため、「すべてのモデルは間違っている、一部は有用である」という格言が適用される。 設計者は、モデルが実際のシステムの重要な側面を捉えていることを確認する必要がある。 それを達成することは特別なスキルであり、その習得には慎重な訓練が必要である。 また、我々は、特定

の問題領域で実際的なメリットを得ることにのみ関心があり、包括的な調査は試みていない。 したがって、その有用性は、他のツールや他の問題領域で異なる場合がある。

参考文献

1. Abrial, J. Formal methods in industry: Achievements, problems, future. In Proceedings of the 28th International Conference on Software Engineering (Shanghai, China, 2006), 761–768.

2. Amazon.com. Supported Operations in DynamoDB: Strongly Consistent Reads. System documentation; http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/APISummary.html

3. Barr, J. Amazon S3: The first trillion objects. Amazon Web Services Blog, June 2012; http://aws.typepad.com/aws/2012/06/amazon-s3-the-first-trillionobjects.html

4. Barr, J. Amazon S3: Two trillion objects, 1.1 million requests per second. Amazon Web Services Blog, Mar. 2013; http://aws.typepad.com/aws/2013/04/amazons3-two-trillion-objects-11-million-requests-second.html

5. Batson, B. and Lamport, L. High-level specifications: Lessons from industry. In Formal Methods for Components and Objects, Lecture Notes in Computer Science Number 2852, F.S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Springer, 2003, 242–262.

6. Bolosky, W., Douceur, J., and Howell, J. The Farsite Project: A retrospective. ACM SIGOPS Operating Systems Review: Systems Work at Microsoft Research 41, 2 (Apr. 2007), 17–26.

7. Brooker, M. Exploring TLA+ with two-phase commit. Personal blog, Jan. 2013; http://brooker.co.za/blog/2013/01/20/two-phase.html

8. Holloway, C. Michael Why you should read accident reports. Presented at the Software and Complex Electronic Hardware Standardization Conference (Norfolk, VA, July 2005); http://klabs.org/richcontent/conferences/faa_nasa_2005/presentations/cmh-whyread-accident-reports.pdf

9. Joshi, R., Lamport, L. et al. Checking cache-coherence protocols with TLA+. Formal Methods in System Design 22, 2 (Mar, 2003) 125–131.

10. Kudrjavets, G., Nagappan, N., and Ball, T. Assessing the relationship between software assertions and code quality: An empirical investigation. In Proceedings of the 17th International Symposium on Software Reliability Engineering (Raleigh, NC, Nov.2006), 204–212.

11. Lamport, L. The TLA Home Page; http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html

12. Lamport, L. Fast Paxos. Distributed Computing 19, 2 (Oct. 2006), 79–103.

13. Lamport, L. The Wildfire Challenge Problem;
http://research.microsoft.com/en-us/um/people/lamport/tla/wildfire-challenge.html

14. Lamport, L. Checking a multithreaded algorithm with +CAL. In Distributed Computing: 20th International
Conference, S. Dolev, Ed. Springer-Verlag, 2006, 11–163.

15. Lamport, L. and Merz, S. Specifying and verifying faulttolerant systems. In Formal Techniques in Real-Time
and Fault-Tolerant Systems, Lecture Notes in Computer Science, Number 863, H. Langmaack, W.-P. de Roever, and J. Vytopil, Eds. Springer-Verlag, Sept. 1994, 41–76.

16. Lamport, L., Sharma, M., Tuttle, M., and Yu, Y. The Wildfire Challenge Problem. Jan. 2001;
http://research.microsoft.com/en-us/um/people/lamport/pubs/wildfire-challenge.pdf

17. Lu, T., Merz, S., and Weidenbach, C. Towards verification of the Pastry Protocol using TLA+. In Proceedings of Joint 13th IFIP WG 6.1 International Conference and 30th IFIP WG 6.1 International Conference Lecture Notes in Computer Science Volume 6722 (Reykjavik, Iceland, June 6–9). Springer-Verlag, 2011, 244 –258.

18. Newcombe, C. Debugging Designs. Presented at the 14th International Workshop on High-Performance Transaction Systems (Monterey, CA, Oct. 2011);
http://hpts.ws/papers/2011/sessions_2011/Debugging.pdf and associated specifications
http://hpts.ws/papers/2011/sessions_2011/amazonbundle.tar.gz

19. Newcombe, C. Why Amazon chose TLA+. In Proceedings of the Fourth International Conference Lecture Notes in Computer Science Volume 8477, Y.A. Ameur and K.-D. Schewe, Eds. (Toulouse, France, June 2–6). Springer, 2014, 25–39.

20. Patterson, D., Fox, A. et al. The Berkeley/Stanford Recovery-Oriented Computing Project. University of California, Berkeley; http://roc.cs.berkeley.edu/

21. Tasiran, S., Yu, Y., Batson, B., and Kreider, S. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In Proceedings of the Third IEEE International Workshop on Microprocessor Test and Verification (Austin, TX, June). IEEE Computer Society, 2002.

22. Zave, P. Using lightweight modeling to understand Chord. ACM SIGCOMM Computer Communication Review 42, 2 (Apr. 2012), 49–57.