リアクティブプログラミングに基づく分散計算基盤の可能性

紙名 哲生
大分大学
kamina@oita-u.ac.jp

要旨

リアクティブプログラミング（*RP*）言語が，実世界に分散配置された様々なモノやサービスを繋げて新たな価値を生み出すための適切な抽象化を提供できるかを考える．とくに，*RP* における時変値を分散システムに適応させる際の整合性維持や永続化などの課題について議論する．

1. はじめに

我が国が目指すべき未来社会の姿として，仮想空間と現実世界を高度に融合させて様々な課題を解決する人間中心の社会 (Society 5.0) が提唱されている．ここでの目標は，ソフトウェアの技術を用いてあらゆる人とモノとが IoT(Internet of Things) 機器等を通じて繋がり，様々な知識や情報サービスの共有によって新たな価値を生み出すことである．ここでいうソフトウェア技術のひとつとして代表的なものにマイクロサービスアーキテクチャがある．これは複数の小さなサービスを API によって連携させ，個々の修正や変更を迅速に行うことができる現在主流の分散アーキテクチャである．

一方こうした連携を実現する際には，個々のサービス間のデータ流の分析が欠かせない．しかしながら，データ流の分析と命令的なプログラミング言語を用いた実装との間には大きなギャップがある．連携をより直観的に記述するには，サービス間のデータ流を宣言的に記述できる抽象度の高いプログラミング言語が必要である．

そのような抽象化を行うプログラミング技術の一つに，**リアクティブプログラミング（RP）**がある．これは環境やユーザからの入力など，時間変化する値に応答するプログラムの記述に適している．とくに Angular，React，Vue などに代表される RP フレームワークは Web アプリケーションのフロントエンドとして成功している．

本稿では，RP 言語で用いられる抽象化である**時変値**を用いて，高い抽象度でのサービス間連携を宣言的に記述できるかについて，またそれを実現する上で解決すべき課題について議論する．

2. 時変値

時変値とはプログラムにおける変数の一種であるが，通常の変数とは異なり，明示的な再代入を行わず，時間とともにその値が変化する[1]．例として，Java を拡張した RP 言語 SignalJ[6] による時変値の宣言を以下に示す．

```
signal int a = 6;
signal int b = Integer.bitCount(a);
a++;
System.out.println(b); // 3
```

SignalJ では時変値の宣言時に修飾子 `signal` を用いる．上の例では二つの時変値 `a` と `b` が宣言されており，`b` は `a` を用いて計算される．`a` の値がインクリメント演算によって更新されると，`b` の値（`a` をビット列としたときの “1” の数）も再計算され，自動的に 2 から 3 に更新される．このように時変値同士を連携させる言語機構は，GUI やセンサとモータの連携など，時間変化する入力列を処理するプログラムの記述に便利である．

実世界に存在するデータ（例えば気象情報，IoT センサ，SNS におけるタイムラインなど）も多くは時間とともに値が変わる．これらを時変値として抽象化し，上の `bitCount` のように何らかのオペレータを用いて宣言的に連携を記述できれば，データ流をそのままプログラム

[1] 本稿では論理的な時間を扱う．ただし，後に示すとおり，永続時変値においてはそれが実時間の時刻印と結び付くことになる．

```
var2 = var1 * 2
var3 = var2 + var3
```

図 1. グリッチの例

として記述することができ，分析とのギャップが小さくなる．果たしてそのようなことは可能であろうか？

3. グリッチと分散 RP

RP 言語が抱える問題の一つに，分散環境におけるグリッチの回避がある．グリッチとは，値更新時に起こる一時的な不整合のことである．例えば，図 1 に示す関係式で成り立つ三つの時変値 var1，var2，var3 があるとする．var2 は var1 を用いて，var3 は var1 と var2 を用いて計算される．いま，var1 の値が 1 に設定されたとする．すると var2 と var3 の値が再計算されるが，もし var3 の値が var2 の値よりも先に計算されてしまうと，まだ確定していない var2 の値を用いて var3 の値が計算されてしまうことになる．

一般的には，こうした不整合はクロックを用いたり [5] 各時変値の計算の順序を正しく決めれば回避できる [10]．しかし，それぞれの時変値が異なる計算ノードで計算される分散環境においてはこの回避は容易ではない．まず，ネットワークの遅延や障害の影響を考慮に入れる必要がある．また，時変値の計算の順序を誰がどのように決定するのかという問題がある．閉じた一つのプログラムにおいては，それはコンパイラが決定すればよかったが，分散システム上においては，特別なコーディネータが決定を行うか（コーディネータで障害が起きた際の影響は大きい）[4]，あるいはノード同士が情報を送りあって計算の順序を決定するような分散アルゴリズムを考える必要がある [11]．

一方で，一般的に整合性の問題は分散システムにおいては古くから議論されていることであり，RP 言語もその議論の上に成立させることが望ましいと考えられる．

4. 時変値の永続化

また，上に示した気象情報などの「時間変化する値」は多くの場合永続データで，過去の値に対する問い合わせを行える場合が多い．一方で RP 言語の時変値は通常一時的な値で計算が終わると消えるため，こうした「時間変化する値」の適切な抽象化にはならない．そのため，**永続時変値** [7] が提案された．これは値変化の履歴を含めて抽象化した時変値のことで，上述した時変値の特徴に加えて，値変化に関する問い合わせを行う API を持っている．内部的には，各履歴の時刻印をキーにした時系列データベースを用いて実装される[2]．

例として，車輌追跡システムにおける以下の車輌クラスを考える．

```
signal class Vehicle {
  persistent signal double x, y;
  signal double v = calcVelocity(x,y);
  Vehicle(String id, double x,
  double y) { init(x,y); ... }
}
```

各車輌は状態として現在の座標値を表す x，y，それに各座標の変化量から求められる推定速度 v を持つ．これらはいずれも時変値である．さらにこれらが永続時変値であることを示す修飾子 persistent が付けられている．ここでは persistent と連携する時変値も永続時変値として扱われるため，v も永続時変値である．車輌クラスをインスタンス化する際には明示的に識別子が与えられ（上のコンストラクタ中の仮引数 id は必須である），その識別子と各 x，y，v の値変化が紐づけられる．ある車輌に対して，例えば指定した時刻における推定速度を求めるには，以下のように snapshot メソッドを用いる（これは上の車輌クラスのように signal と修飾されたクラスが暗黙的に持つ API メソッドである）．

```
double sampleV =
  new Vehicle("Oita_a_1234")
  .snapshot("2018-06-01T18:10:00").v;
```

まず，車輌クラスのインスタンス生成に識別子として"Oita_a_1234"を与えることにより，その識別子と紐づいた時変値の集まり（x, y, v）を得る．次に snapshot メソッドを用いて時間を遡り，フィールド v にアクセスすることでその時点での推定速度を得る．

上で生成されたインスタンスが消滅したり，アプリケーションそのものが停止したとしても，時変値が持つ履歴は，明示的に破壊されない限りディスクに残り続ける．永続時変値では，このように通常の時変値とは異な

[2]履歴は，理想的には全てを残しておけるのが望ましいが，それが現実的でない場合は不要になったデータを捨てることになる．そのような管理も，データベースを前提にすれば容易になる．

るライフサイクル管理の機構が用意されている．また，永続時変値は，センサとモータの連携のような閉じたアプリケーションだけではなく，オープンな時系列データとして提供されることも想定している [8].

5. 永続時変値と分散 RP

ところで，分散 RP におけるグリッチやその他の整合性の問題は，永続時変値においては分散時系列データベースにおける整合性の問題であると言い替えることができる．まず，オープンにアクセスされる場面を想定すると，永続時変値にアクセスする複数のプロセス間に矛盾があってはならない．一方，永続時変値においては全ての更新履歴に時刻印が付けられるため，時刻印の存在を前提にできる．時刻印の付けられ方が実世界で発生した事象の順序を忠実に再現することを要求するのか，何らかの因果関係との間に矛盾がなければよいのか [1]，あるいは観測された結果において矛盾がなければよいのか [2] については，アプリケーションに依存するであろう．また時刻印についても，大域的な時刻を想定するのか，Lamport の時刻印 [9] のようなものを用いるかなど，様々であろう．いずれにせよこうした問題については古くから数多くの研究がなされている．

グリッチの回避については，例えば関連する全ての永続時変値の値を先に計算しておき，それぞれに適切な時刻印（例えば共通する時刻印）を付与して分散したデータベースを同時更新する方法がある．図 1 の例の場合，予め `var1`，`var2`，`var3` の値を計算しておく（単一のノードで計算するのでグリッチは容易に回避できる）．次にそれらの値に同じ時刻印を適切に付与し，各ノード上のデータベースに格納する．分散コミット [3] の仕組みで各更新が成功しなければ観測されることはなく，不整合は生じない．永続時変値はこのような古くからの分散システムの分野における議論の上に成立しているため，分散 RP におけるグリッチ問題に対する解は既に与えられている場合が多そうだと予測できる．

6. まとめと今後の課題

RP 言語の抽象化である時変値を用いて宣言的なサービス間の連携が可能かどうかについて，とくに分散 RP におけるグリッチと時変値の永続化の問題を取り上げて議論した．永続データを含む「時間変化する値」は永続時変値を用いて RP 言語上で適切に表現することができ，またそれによりグリッチ問題も分散システムにおける整合性の問題へと帰着させることができる．永続時変値の研究は始まったばかりであり，それを用いた具体的なサービス間連携の実証が行われているわけではないものの，それが Society 5.0 を支えるソフトウェア開発のための便利な言語要素となる可能性は十分にある．

最後に，永続時変値を実用的な広域分散計算基盤として使える言語要素にしていくために今後取り組むべき課題について議論する．

ID 管理： 永続時変値において，インスタンス生成時に与える識別子はライフサイクル管理や個々の時変値の取得に用いられる必須のものである．識別子はプログラマが自由につけられるが，名前の衝突があってはならない．とくに広域的な計算基盤として永続時変値を用いるには，名前の衝突回避のための ID 管理機構を考える必要がある．

ソフトウェア進化： 永続時変値を内部的に管理する時系列データベースは，永続時変値をメンバに持つクラス（例えば上述の車輌クラス）の宣言を反映している．クラス定義に変更があれば，時系列データベースも同時に変更されるべきであり，その際に不整合が生じないようなデータベース変更の機構を考える必要がある．

参考文献

[1] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[2] Sebastian Burckhardt. Principles of eventual consistency. 2014.

[3] Eric C. Cooper. Analysis of distributed commit protocols. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, pages 175—-183, 1982.

[4] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed REScala: an update algorithm for distributed reactive programming. In *OOPSLA'14*, pages 361–376, 2014.

[5] Nicholas Halbwachs, Paul Caspi, Pascal Paymond, and Daniel Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[6] Tetsuo Kamina and Tomoyuki Aotani. Harmonizing signals and events with a lightweight extension to Java. *The Art, Science, and Engineering of Programming*, 2(3), 2018.

[7] Tetsuo Kamina and Tomoyuki Aotani. An approach for persistent time-varying values. In *Onward!'19*, pages 17–31, 2019.

[8] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Signal classes: A mechanism for building synchronous and persistent signal networks. In *ECOOP 2021*, volume 194 of *LIPIcs*, pages 19:1–19:30, 2021.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558—565, 1978.

[10] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA'09)*, pages 1–20, 2009.

[11] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. Distributed reactive programming for reactive distributed systems. *The Art, Science, and Engineering of Programming*, 3(3):5:1–5:52, 2019.