小規模組み込みシステム向けFRP言語のアクターモデルにもとづく実行系

渡部卓雄（東京工業大学 情報理工学院）

マイクロコントローラ等の小規模組み込みシステム向けに設計された関数リアクティブプログラミング(FRP)言語のためのアクターモデルにもとづく実行方式を提案する．提案方式では，FRP言語の特徴である時変値をアクターで，時変値間の値伝搬を非同期メッセージでそれぞれ表現する．本実行方式により，小規模システムでの実行を考慮して静的に実現されている言語の実行系にある程度の柔軟性を与えることが可能になる．

関数リアクティブプログラミング(FRP)

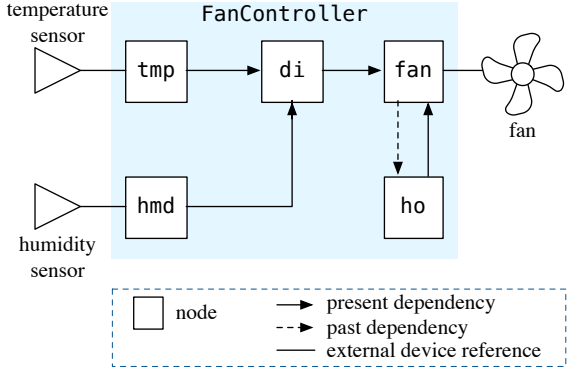
関数リアクティブプログラミング(FRP)は，時間と共に変化する値を表す時変値(time-varying value)やイベントストリーム等の抽象化機構を用いて入力やそれらに依存する値を表現し，リアクティブシステムの宣言的な記述を支援するプログラミングパラダイムである．
時変値はシグナルとも呼ばれる．型Tの時変値の型 Signal T は，概念上時刻から型Tへの関数の型 Time → T で表すことができる．しかしTimeの値をプログラム中に明示的に導入すると時間・空間漏洩等の不都合が生じるため，通常はこれを隠蔽した形で時変値を表す．そのための手法としてはシグナル関数(SF T T' = Signal T → Signal T')の導入等がある．本研究で使用する言語Emfrpでは，Elmと同様に Signal T を組込み型とすることでTimeを隠蔽している．加えて，時変値を一級データとはせず，かつ時変値の時変値は扱わないといった制約を設けることで，リソース制約の厳しい小規模組込みシステムでの実現を可能にしている．

```
module FanController  # module name
in  tmp : Float,      # temperature sensor
    hmd : Float       # humidity sensor
out fan : Bool        # fan switch
use Std               # standard library

# discomfort (temperature-humidity) index
node di = 0.81 * tmp + 0.01 * hmd
          * (0.99 * tmp - 14.3) + 46.3

# fan switch
node init[False] fan = di >= 75.0 + ho

# hysteresis offset
node ho = if fan@last then -0.5 else 0.5
```

EmFrpによるファンのコントローラ

K. Sawada & T. Watanabe, "Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems", CROW 2016

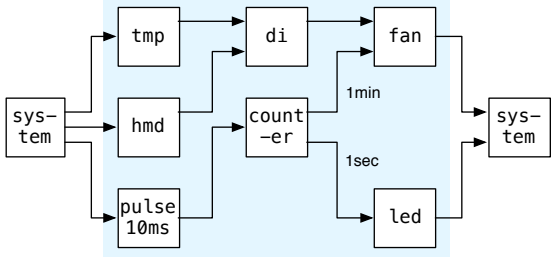
FRP言語の実行方式

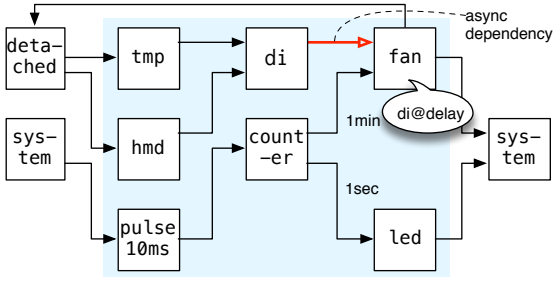


FRP言語によるプログラムは，時変値をノードとし，それらの依存関係を辺とした有向非巡回グラフ(DAG)で表現できる．プログラムの実行は，依存関係に沿ってノードの値を順次更新することの繰り返しとして実現される．左図の場合，例えば tmp, hmd, di, ho, fan の順にノードの値を更新する．この実行方式はpush型と呼ばれる．これに対してpull型の実行方式では，出力に相当するノード（左図の例ではfan）が依存するノードに更新要求を出し，それが入力に相当するノード（例ではtmpとhmd）に至るまで順次伝搬する．
push型の実行方式は少ないリソースで実現可能であり，かつインターバルタイマーによる周期的タスクの表現などに適している．その一方で，pull型の実行では生じないような無駄な計算（不要なノードの更新）が発生することがある．本研究では，時変値をアクター，時変値間の値伝搬をメッセージで表現することで，push型をベースとした実行にpull型を混在させる等，さまざまな実行方式を可能にする手法を提案する．

アクターにもとづく実行方式

本方式では時変値をアクター，時変値間の値伝搬をメッセージで表現する．右上図はタイマーを用いたファン制御を表している．この実行方式では，systemアクターが入力ノード（アクター）にメッセージを送信し，最終的に出力ノード（アクター）からsystemにメッセージが到達するというサイクルを繰り返す．この例では1分に一回だけdiの値をチェックしてファンのON/OFFを行い，かつ1秒毎にLEDを点滅させる．この例をpush型で実行するとtmp，hmd，diの更新のほとんどが無駄になる．特にtmpとhmdはセンサをアクセスするため余分な電力を消費する．しかしpull型の実行ではledのような周期的タスクに対応できない．
そこで右下図のように変更する．まずdiからfanへの値伝搬を非同期的にする．具体的にはfanは通常counterからのメッセージのみを受信し，diからは必要な時のみメッセージを受信するようにする．これだけでは上で述べた無駄な更新は解消できないため，diが依存している全ノードのsystemからの依存を外し，代わりにdetachedに依存させる．そしてfanは必要に応じてdetachedに要求メッセージを送信する．これによってtmp，hmd，diの更新は必要な場合のみ行われるようになる．以上はコンパイラに@delayディレクティブを導入し，di@delayとすることで自動的に行えるようにする．この例のようにメッセージの送信順序を制御することで種々の柔軟な実行方式を導入できる．





タイマーによるファン制御

```
module FanController # module name
in  tmp : Float,     # temperature sensor
    hmd : Float      # humidity sensor
    pulse10ms : Bool # hardware interval timer (10ms)
out fan : Bool,      # fan switch
    led : Bool       # LED
use Std              # standard library

# discomfort (temperature-humidity) index
node di = 0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3

# timer counter (resets to 0 every 1 minute)
node init[0] counter =
     if !pulse10ms@last && pulse10ms   # detect rising edge
     then (counter@last + 1) % 6000 else counter@last

# fan switch
node fan =
     if counter@last != counter && counter == 0  # detect 0 reset
     then di >= 75.0 else fan@last

# LED blinks at 1Hz
node led = (timer % 100) < 50
```

di@delay

このように変更することでdiおよびdiが依存しているtmpおよびhmdの更新がオンデマンドになる