WSAN向けマクロプログラミング言語の提案

後藤 司　森口 草介　渡部 卓雄

無線センサーアクターネットワーク (WSAN) は，物理環境に配備された複数個の計算機が相互に通信することで，物理環境の観測や環境への働きかけを行う分散システムである．その実現には，センサーノードに加えて環境への働きかけなどを行うアクターノードを含む複雑なノード間協調が必要である．ノード間協調が不十分な場合，センサーやアクターへのメッセージの到達順序が入れ替わり，実行順序が意図しないものになるハザードと呼ばれる問題が発生することがある．アクターノードを含まない無線センサーネットワーク (WSN) については，マクロプログラミングと呼ばれる，WSN 全体を 1 つの計算システムとみなしてその動作を記述する手法がある．マクロプログラミングでは処理に用いるノード間協調を全体の動作から導出する．本研究では，ハザード回避を行う協調動作を導出する機構を取り入れた WSAN 向けのマクロプログラミング言語を提案する．

Wireless Sensor and Actor Networks (WSANs) are distributed systems composed of multiple computers that observe and act on physical environments. Since WSANs contain actor nodes that control actuators or perform other local operations, complex inter-node coordination operations involving them are necessary. The lack of such coordination may result in the incorrect arrival order of messages between actor and sensor nodes. The problems are called hazards. For wireless sensor networks (WSNs), which do not have actor nodes, there is a method called macro-programming, which describes the behavior of the entire WSN as a single computing system. In macro-programming, inter-node cooperation for processing is derived from the overall behavior. In this study, we propose a macro-programming language for WSAN that incorporates a mechanism to derive cooperative actions to avoid hazards.

1　はじめに

無線センサーネットワーク (Wireless Sensor Network, WSN) は，センサーと無線通信機構を備えた複数個の計算機を用いて多地点の計測情報を収集・活用するための分散システムであり，農業，工業，建築，土木，交通，医療などの分野で用いられている．WSN を構成する計算機（ノード）は，センサーによる計測とパケットのルーティングを行う複数個の**センサーノード**と情報の集約を行う単一の**シンクノード**に分類される．WSN に**アクターノード**と呼ばれる計算機を追加したものが**無線センサーアクターネットワーク (Wireless Sensor and Actor Network, WSAN)** である [1]．アクターノードは環境に何らかの影響を及ぼすアクチュエーター（ヒーター，ファン，モーター等）の制御や，近傍にあるセンサーノードからのデータの集約，近傍にあるセンサーノードの制御等を行う．

WSAN では，センサによる計測とアクチュエータの制御を適切に行うためのノード間協調が必要になる．ノード間協調における問題の 1 つとして，シンクから各ノードへの命令の実行順序が入れ替わってしまう**ハザード** [6] と呼ばれる現象がある．以下，火災を検知してスプリンクラーを制御する WSAN アプリケーションを例として説明する．このアプリケーションでは，温度センサーの値を取得するクエリと，温度センサーの値が閾値より高い場合に火事と判断してスプリンクラーを起動するコマンドの 2 種類の命令をシンクが送信する．火事が消える瞬間にシンクが送信したクエリとコマンドの到達順序が入れ替わってし

A Macroprogramming Language for Wireless Sensor-Actor Networks

Tsukasa Gotoh, Sosuke Moriguchi, Takuo Watanabe, 東京工業大学 情報理工学院, School of Computing, Tokyo Institute of Technology.

まうと，シンクはまだ火災が発生していると判断して余計にスプリンクラーを起動して必要以上に水を散布してしまう．このような協調に対する問題があることから WSAN の開発は WSN より複雑になる．

ハザードを回避する手法として，火災などのイベントが発生している領域内にあるセンサーノード，アクターノード同士で命令実行順序を同期するための **Vector Neighborhood Clock (VNC)** による手法 [6] が提案されている．VNC ではクエリやコマンドの実行を制御するために近くのノード間で通信を行う．この通信は本来のクエリやコマンドと並行して行われるため，それぞれのノード用プログラムではクエリ・コマンド用のコードと VNC のためのコードが混在してしまう．そのため，プログラム間の整合性を保つことが難しく，また機能に対する再利用性が悪化する．

WSN のためのソフトウェアを効果的に開発する技術として**マクロプログラミング** [5] と呼ばれる手法がある．これは，個々のノード単位ではなくグローバルな単位でネットワークをプログラミングするようなプログラミングパラダイムであり， コンパイラはグローバルな単位のプログラムをノード単位のプログラムに変換する．

本研究では，WSAN 開発のためのマクロプログラミング言語 Brigade を提案する．Brigade では各ノードの状態を時変値とし，ノードの種類ごとにまとめた時変値のストリームを処理してプログラミングを行う． 1 つの Brigade プログラムからセンサーノード，アクターノード，シンクノードの 3 種類の異なるノードに対するプログラムが生成される．このストリームによる抽象化によって WSAN アプリケーションのプログラムを簡潔に記述することができる．さらに各プログラムの生成を行う過程で VNC による協調動作が組み込まれ，ノード間協調の問題の 1 つであるハザードをプログラマが意識することなく回避することが可能となる．

本論文ではまず第 2 章で WSAN について説明し，第 3 章で VNC について述べる．第 4 章で Brigade 言語について説明し，第 5 章では Brigade の実装手法を説明する．第 6 章で関連研究の紹介を行い，第 7 章で結論と今後の課題について述べる．

2 無線センサーアクターネットワーク

2.1 WSAN の概要

無線センサーネットワーク (Wireless Sensor Network, WSN) はセンサーや無線通信機能を備えた計算機を複数個物理環境に配備することで，環境の観測データを収集するための分散システムである．WSN において計算機はノードと呼ばれ，センサーで物理環境を計測するためのノードをセンサーノード，情報の収集を行うノードをシンクノードと呼ぶ．通常 WSN は多数のセンサーノードと 1 つのシンクノードから構成され，センサーノードにより計測されたデータを無線通信によりセンサーノードを経由してシンクノードへ送信する．シンクノードに集約されたデータは記憶領域に保存されたり，ネットワーク経由で参照されたりする．

無線センサーアクターネットワーク (Wireless Sensor and Actor Network, WSAN) は，WSN にアクターノードと呼ばれるノードを追加したものである．アクターノードはセンサーノードによって集めた物理環境の情報を元に適切に物理装置を動作させることで，物理環境に働きかけることを目的としている．センサーノードは消費電力が小さく，センサーの観測範囲や計算資源，通信機能が制限されていることに対して，アクターノードの持つ物理装置はセンサーノードの観測範囲よりも広い影響範囲を持ち，計算能力と通信能力も高くなる．このため，センサーノードよりもアクターノードのほうが高価になり，配備される数は少なくなる．

2.2 ハザード

ハザード [6] はセンサー，アクター，シンク間の協調動作の不足によって，物理環境に望ましくない変更を与えてしまうような命令の実行順序の入れ替わりの事である．シンクが正しい順序で命令を送信していても，到達パスの長さや，レイテンシの違いによって実行順序が入れ替わる．以下ではシンクがアクターに送信する命令をコマンド，センサーに送信する命令をクエリと呼ぶ．

火災が発生している領域のように，何らかの事象が発生している領域のことを**イベント領域**と呼ぶ．イベント領域の情報をセンサーで読み取ったり，イベント領域に対してアクターを用いて働きかけるといった操作が複数回必要となる WSAN アプリケーションを考える．

図 1 は WSAN におけるノードの配置とイベント領域を表した図である．大きな円はイベント領域を，小さな円，三角形，四角形はそれぞれセンサー，アクター，シンクを表す．C_1, C_2 はアクターへのコマンドであり，Q_1, Q_2 はセンサーへのクエリである．R_1, R_2 はクエリ Q_1, Q_2 に対するレスポンスとなっている．C_1 と Q_1 は C_2 と Q_2 よりも先に送信され，長いパスを通ることで C_2 と Q_2 よりもイベント領域までの到達に時間がかかるものとしている．

[6] では 2 つの命令の種類とその前後関係に応じてハザードを CAC, QAC, CAQ, QAQ の 4 種類に分類している．例えば QAC は Query-after-Command の頭字語であり，C_1, Q_2 の順序でコマンド C_1 とクエリ Q_2 を送信したときに，C_1 の実行終了時刻よりも先に Q_2 の実行が開始されるハザードである．シンクが消火コマンド C_1 を発行し消火を行った後に火災を検知するクエリ Q_2 を発行する際に，消火が終了する前に火災検知が行われてしまうと，火災が鎮火したとしてもシンクは火災が発生していると判断して余計に消火コマンドを発行してしまう．他にも，C_1, C_2 が入れ替わるのが CAC ハザード，Q_1 と C_2 が入れ替わるのが CAQ ハザードである．QAQ ハザードは Q_1 と Q_2 が正しい順序でイベント領域まで到達し，レスポンス R_1, R_2 が入れ替わる際に発生する．

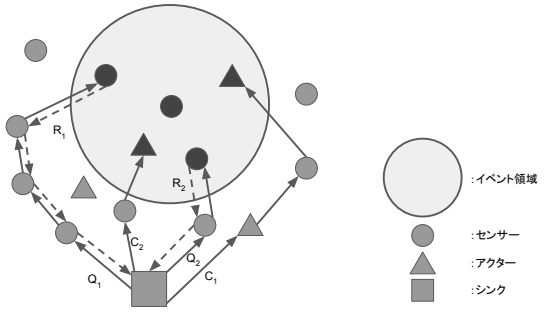

図 1　WSAN

2.3　マクロプログラミング

マクロプログラミングは，WSN をグローバルな単位でプログラミングをするようなプログラミングパラダイムである．マクロプログラミング言語で記述されたグローバル単位のプログラムは，コンパイラによってノード単位のプログラムに変換される [5]．ノード単位のプログラミングをする上で意識する必要のある，ハードウェアに近い低レベルな部分や通信のプログラムを自動で生成することによって，開発効率を高める事ができる．

Regiment [5] は関数型リアクティブプログラミングに基づく WSN 向けのマクロプログラミング言語である．ネットワークを空間的に分散した時変値の集合とみなす．時変値は各センサーノードの状態や，物理的な近傍の情報を集約した情報などである．これら時変値やその集合を操作できるプリミティブ関数を用いてプログラミングを行い，専用の中間言語である Token Machine Language (TML) [4] へとコンパイルを行う．

このような WSN マクロプログラミング言語を WSAN に適用する際の問題点として，アクターノードを追加することで現れるノード間協調動作への対処となる．本研究では特にハザードに注目し，ハザード回避を自動で行う機構を持つ WSAN 向けマクロプログラミング言語を提供する．

3　Vector Neighborhood Clock

3.1　Neighborhood Clock

ハザードを回避する手法として Neighborhood Clock (NC) が提案されている．NC は特定の範囲内に限定された仮想クロックによる同期手法である．

あるノード x の近くのノードを包含し，そのノードと x 間で命令が順序どおりに実行される必要のある領域を**依存領域**という．観測範囲が重なっているセンサーや影響範囲が重なっているアクター，観測範囲と影響範囲が重なっているセンサーとアクターは互

いの依存領域内に存在している．ここではセンサーの依存領域の半径はセンサーの観測半径とアクターの影響範囲の半径の和で，アクターの依存領域の半径はアクターの影響範囲の半径の 2 倍で定義される．依存領域に対する 2 つの命令 I_i, I_j に対し，依存領域内のあるノードが I_j を実行する前に依存領域内の全てのノードが I_i を実行する必要があるとき，I_j は I_i に依存していると言い，$I_i \rightarrow I_j$ と書く．

シンクが送信する命令 I_i, I_{i+1} に対して $I_i \rightarrow I_{i+1}$ であるとする．全てのセンサーとアクターは NC というスカラー値を持つ．シンクは命令の依存の進行具合を示す参照クロック RC を生成し，その初期値 NC_0 をイベント領域全体に送信する．イベント領域内のセンサーとアクターは自身の NC 値を NC_0 で初期化する．シンクが送信する命令 I_i に RC_i 値を乗せ，命令を送信するたびに RC 値をインクリメントする．センサー及びアクターは命令 I_i の実行が完了したら依存領域内の全ノードに自身の NC 値を送信する．センサーに対するコマンドのように，対象が自身でない命令に関しては届いたらすぐ実行完了通知を送信する．各ノードは依存領域内のノードの NC 値を保持しており，全てのノードの NC 値が NC_i になった時に初めて I_{i+1} を実行できるようになる．

これにより I_i の実行が完了する前に I_{i+1} が実行されなくなり，CAC，QAC，CAQ ハザードが回避できる．また QAQ ハザードはクエリのレスポンスの到達時刻が入れ替わった際に発生するハザードであるが，レスポンスに RC 値を乗せることでシンクがレスポンスの順序を判別することができるため，QAQ ハザードも回避できる．

3.2 Vector Neighborhood Clock

複数種類のイベントが発生していた場合は，イベント毎に命令の依存関係を定義し，関係のないイベントの命令の実行は待たないようにしたい．Vector Neighborhood Clock (VNC) はイベントごとの命令の依存関係に基づき NC を持つようにしたものである．イベントが k 種類定義され，イベント毎の依存関係リストが最大 j 個ある場合，VNC は $k \times j$ 次元となる．ただし，Brigade 言語では各イベント毎の全命令は同期させるため依存関係リストは 1 つとなる．そのため本研究で扱う VNC は k 次元となる．各次元に対して NC の手法を用いることでイベント毎の同期が可能となる．また同じイベントが離れた位置で発生していても区別せず，同じイベント ID を割り当てる．

例として 2 種類のイベント A, B と，イベント A の命令 A_1, A_2 及びイベント B の命令 B_1, B_2, B_3 を考え，それぞれ $A_1 \rightarrow A_2$, $B_1 \rightarrow B_2 \rightarrow B_3$ という依存関係を持つとする．シンクは RC 値を $\{NC_A : 0, NC_B : 0\}$ で初期化してイベント領域に送信し，各センサーとアクターの VNC 値は $\{NC_A : 0, NC_B : 0\}$ になる．シンクが A_1 を送信するとき，シンクの保持する RC の値 NC_A をインクリメントして 1 にし，それをイベント A 領域のノードに送信する．イベント A 領域内のノードは，依存領域内のノードの NC_A 値が 0 であるため，A_1 を実行できる．実行が完了したら自身の NC_A 値を依存領域内の全ノードに送信する．B_1 を送信するときも同様に，NC_B をインクリメントして 1 にして送信し，イベント B 領域内のノードが B_1 を実行する．A_2 が送信されるとき，依存領域内のノードが NC_A 値をインクリメントして 2 にして送信する．イベント A 領域内のノードは，依存領域内のノードの NC_A 値が全て 2 であれば A_2 を実行し，そうでなければ他ノードから NC_A 値が送信されるのを待ってから実行する．同様にして表 1 のように実行が行われる．

表 1　VNC 実行例

命令	RC
A_1	$\{NC_A : 1, NC_B : 0\}$
B_1	$\{NC_A : 1, NC_B : 1\}$
A_2	$\{NC_A : 2, NC_B : 1\}$
B_2	$\{NC_A : 2, NC_B : 2\}$
B_3	$\{NC_A : 2, NC_B : 3\}$

4 Brigade 言語

4.1 言語の概要

Brigade は本研究で提案する WSAN 向けの関数型マクロプログラミング言語である．各ノードの状態を時間によって変化する値（時変値）として抽象化し，時変値の集合をストリームとする．例えば，センサーが観測した値やその値を変換した情報などの集合がストリームとなる．時変値は概念的には時刻から値への関数であるが，Brigade ではサンプリングされた現時刻における値に対して処理を行う．ストリームに関数を適用することで値の変換を行い，あるノードのストリームを他ノードが参照することでノード間のストリームデータの送信が行われる．センサーからシンク，シンクからアクターへとストリームを参照することで WSAN アプリケーションを構成する．1 つの Brigade プログラムをコンパイルすることでセンサー，アクター，シンクそれぞれのプログラムが生成される．

また Brigade によって生成されたプログラムは VNC によるハザード回避を自動で行う．シンクが参照するセンサーのストリームの領域をイベント領域として認識し，VNC による同期の対象とする．同領域が対象となるストリームは全て VNC によって同期される．ストリーム毎に同期が取られるため，プログラマは送信された命令がイベント領域単位では同時に実行されないことを意識してプログラムを書く必要がある．

4.2 ストリーム

ストリームは時変値の集合である．Brigade ではストリームは `Stream(t)`型の値として扱われる．`t`にはセンサー ID を表す `Sensor`型や整数型などが入り，特定の関数によってストリーム内の各時刻における `t`型の値を処理し，新たなストリームを生成することで変換が行われる．

`map`関数はストリーム内の各 `t`型の値に関数を適用し，その結果で新しいストリームを構成して返す．以下のプログラムは摂氏温度のストリームを華氏温度のストリームに変換している．

```
map(fn(c) -> c * 9 / 5 + 32, celsiusStream)
```

`filter`関数はストリームの各値に述語を適用し，条件を満たした値のみで構成されるストリームを返す．以下のプログラムは温度のストリームの各値のうち，60 度以上の値のみを含むストリームに変換している．

```
filter(fn(t) -> t >= 60, temperatureStream)
```

センサーやアクターの ID は `Sensor`型と `Actor`型の値によって表される．`sensors`は `Stream(Sensor)`型のストリームであり，一定時間間隔ごとに全センサーの ID をストリームに流す．Brigade プログラム中の全てのストリームの起点は `sensors`であり，この`sensors`に対して関数を適用することによってプログラムを構成する．

各センサーが計測した物理環境の情報の取得やアクターが保持する物理装置の動作の制御は，`sense`関数，及び `actuate`関数によって行われる．`sense`関数はセンサーの ID とセンサーの種類を指定し，対応するセンサーノードによって計測された値を返す．`actuate`関数はアクターの ID と物理装置の種類，及び物理装置を制御するための値によって，対応するアクターの物理装置を動作させる．`sense`関数と `actuate`関数の実装は Brigade 外部で行われ，Brigade ではそれを呼び出す形式になっている．

`actuate`関数の呼び出しは `iter`関数内で行われる．`iter`関数はストリームの各値に対して引数に与えられた関数を適用する．`map`や `filter`と異なり，`iter`は副作用を期待するもので，ストリームを返さない．以下は各アクターのファンに対して値を渡し，ファンを制御している．

```
iter(fn((actorId, switch)) -> actuate(
     actorId, "fan", switch), switchStream)
```

4.3 ブロック

センサー，アクター及びシンクのプログラムはそれぞれ `Sensor`, `Actor`及び `Sink`ブロック内に記述される．例えば `Sensor`ブロック内に記述されたストリームの処理は，コンパイル後はセンサーのプログラム内

に含まれる．ブロックは次のように記述される．

```
Sensor -> { ... }
```

ブロック内で定義された変数のスコープはブロック内に限られるが，ストリーム変数はブロック間での受け渡しが行われる．ブロック間におけるストリーム変数の受け渡しが行われると，対応するノード間で通信が行われる．変数の受け渡しは Sensor・Sinkブロック間，及び Actor・Sinkブロック間で行うことができ，Sensor・Actor間での受け渡しは行うことができない．Brigade 内部では sensorsストリームは Sinkブロック上で定義されたストリームであり，Sensorブロックで sensorsを利用するときは Sinkブロック上の変数を参照することになる．

sensorsストリームは全センサー ID のストリームであり，各要素は対応するセンサーに紐付けられているため，そのままストリームを Actorブロックに渡すことはできない．getActors関数は，Stream(Sensor)から Actorブロックが受け取れる Stream(Actor)型への関数である．この関数は Sinkブロック内でのみ使うことができ，シンクが保持しているセンサーとアクターの位置情報を用いて，ストリーム内のセンサーの依存領域内にある全てのアクターノードの ID をストリームにして返す．

まとめると各関数や値の型は以下の通りとなる．

```
map:(a → b)→Stream(a)→Stream(b)
filter:(a → bool)→Stream(a)→Stream(a)
iter:(a →())→Stream(a)→ ()
sensors:Stream(Sensor)
getActors:Stream(Sensor)→Stream(Actor)
sense:Sensor → string → Int
actuate:Actor → string → Int →()
```

4.4 VNC によるノード間協調

Brigade ではシンクから送信されるデータを命令とみなし，VNC による協調の対象とする．プログラム実行時にセンサーからシンクへストリームのデータが送信されたら，シンクはそのデータを送信したセンサーの観測範囲全体をイベント領域とみなし，そのストリームを変換して構成されたストリームも同イベントとして扱う．

例えば Sensorブロック上の温度の高いセンサー ID のストリームを Sinkブロック上で参照し，スプリンクラーを起動するアクターの ID を同ストリームを変換して構成し，Actorブロック上でそのストリームを参照するとする．最初に Sensorブロックが Sinkブロック上で定義された sensorsストリームを参照しているため，シンクからセンサーへセンサー ID が送信される．この送信を命令とみなして I_1 とし，シンクからアクターへのアクター ID の送信を I_2 とすると，この 2 つの命令は $I_1 \rightarrow I_2$ という依存関係を持つ．さらにアクターにアクター ID を送信した後に再度センサー ID を送信するときには $I_2 \rightarrow I_1$ という依存関係を持つことになる．ただし再度センサーに ID を送信するときに VNC による同期が取られるイベント領域は，全センサーではなく前回センサーからシンクへ送信されたときと同じ領域になり，再度センサーからシンクへ温度の高いセンサー ID が送信されたときにイベント領域が更新される．

VNC による同期に対してプログラマが追加で記述する必要のあるプログラムは無く，上記の機構によりイベント領域毎のハザード回避は全て自動で行われるようになる．

Sensorブロック上の sensorsストリームから分岐した複数のストリームを Sinkブロックが参照している場合は，それぞれがイベント領域とみなされ，VNC の要素に対応付けられる．

4.5 例題

プログラム 1 は Brigade で記述された火災消火プログラムである．60 ℃以上の温度を検知したセンサーの ID をシンクが受け取り，同領域の対応するアクターのスプリンクラーを起動する．

プログラムは関数定義と Sensor, Sink, Actorブロックにそれぞれ分かれており，各ブロック内での計算がそれぞれ対応するノード上で行われる．onFire関数は火事かどうかを判定する関数で，センサー ID の示すセンサーの観測した温度が 60 ℃以上であるかを

どうかを bool値で返す．actuateSprinkler関数はスプリンクラーを起動するための関数であり，対応するアクター ID の持つスプリンクラーを起動している．actuate関数はここでは 1 を渡されたら一定時間スプリンクラーを起動し，一定時間が経過したら停止するように実装されているものとする．Sensorブロックではまず sensorsストリームのセンサー ID を onFire関数で火事と判定されたセンサー ID を抽出したストリームに変換する．それを fireRegionに代入することで Sinkブロック内で参照できるようにし，getActors関数を用いて同じ領域内のアクターのストリーム actorsを構成する．Actorブロックは actorsを参照し，actors内の全てのアクター内でスプリンクラーを起動する．

VNC による協調がない場合，この例では actuateSprinklerの実行が完了する前に onFireが呼ばれてしまい，QAC ハザードが発生する．この WSAN では Sinkブロック上の変数 actorsと sensorsを，それぞれ Actorブロックと Sensorブロックが参照しており，それぞれのデータのシンクからの送信が依存するため，VNC によってアクターが ID を受け取ってから実行を行う actuateSprinklerが完了するまで，センサーが ID を受け取っても onFire関数は実行されない．

```
fn onFire(sensorId) -> {
  sense(sensorId, "temperature") >= 60
}
fn actuateSprinkler(actorId) -> {
  actuate(actorId, "sprinkler", 1)
}
Sensor -> {
  fireRegion = filter(onFire, sensors)
}
Sink -> {
  actors = getActors(fireRegion)
}
Actor -> {
  iter(actuateSprinkler, actors)
}
```

プログラム 1　Brigade による火災消火プログラム

5　実装手法

Brigade コンパイラのコンパイルの手順は図 2 にて表される．字句解析，構文解析，型検査が行われた後にストリームグラフマップを構成し，ブロック間参照表を構成して通信を行うストリームを認識する．参照情報を元に，ストリームをセンサー，アクター，シンク用のプログラムへと変換していく．

5.1　ストリームグラフマップとブロック間参照表

ストリームのブロック間の参照関係を確認したり，センサーやアクターのプログラムの生成を容易にするため，各ブロック内のプログラムからストリームの処理の流れを表す有向グラフのマップと，ストリーム変数のブロック間の参照の表を構成する．

ストリームグラフでは Stream型の変数や，mapや filterなどの Streamを引数に取る関数がノードとなり，関数を適用される側のノードからする側のノードの向きにエッジがつけられる．Stream型の値が変数へ代入されていたら，変数名を key としてストリームグラフをマップに保存する．このマップを後のブロッ

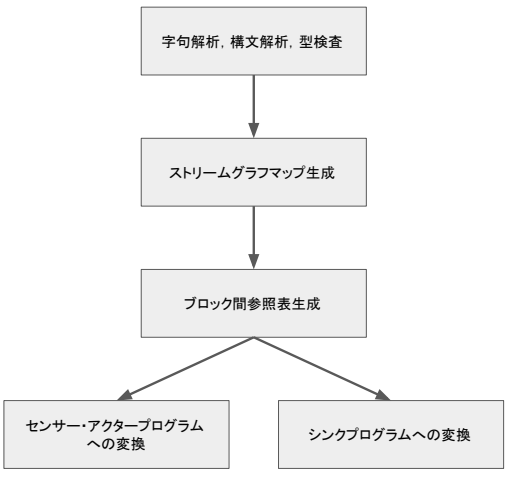

図 2　コンパイル手順

ク間参照表の構成や，センサーやアクターのプログラムの生成に利用する．

Stream型の変数をノードにする際，同ブロック内で定義されていた場合にはマップからストリームグラフを取得して置き換える．同ブロック内で定義されていない場合は他ノードで定義されているものとして変数名がそのままグラフのノードとなる．他のブロックのストリームグラフを繋げてしまうとブロックごとのプログラムの変換が行えなくなるため，他のブロックの変数を参照していてもストリームグラフの置き換えは行わない．

以下の Brigade プログラムを考える．

```
X = map(f1, filter(f2, Y))
```

この時，=の左辺の Xはマップの key となり，右辺の式はストリームグラフへ変換された後にマップの value となる．

$$\begin{aligned} key &= X \\ value &= Y \rightarrow filter(f_2) \rightarrow map(f_1) \end{aligned}$$

図 3　ストリームグラフ変換

また actuate関数の呼び出しなどの副作用を期待している iter関数は変数への代入をしないため，このままではストリームグラフマップへの保存が行われずコード変換が行われない．iter関数の呼び出しだけは変数に代入されているものとして扱い，ストリームグラフマップへと保存する．

上記の形式でセンサー，アクター，シンクの各ブロックのストリームグラフマップを構成したら，ブロック間を跨ぐ Stream型変数の参照をノード間通信に変換するため，ブロック間の変数参照の表を作成して通信が行われる箇所を明らかにする．マップの key に現れる変数を出力，value のストリームグラフの末端に現れる変数を入力として，対応するブロック名と共に表に記録する．出力と入力片方にしか現れない変数は通信が行われないため表から削除するが，iter関数が代入されている変数だけは出力にならないため，削除を行わない．

ブロック間参照表を生成したら各ノード向けのプログラムを生成する．センサーとアクターはそれぞれが単一の値を 1 台のシンクと送受信するのに対して，シンクは複数台のノードと値を送受信するため，プログラムの生成方式が異なる．

5.2　センサー・アクター向けプログラムへの変換

センサーとアクター向けプログラムは，ブロック間参照表を見て自身が出力として記録されている変数が key となるストリームグラフをマップから取得し，そのストリームグラフをイベント駆動型のプログラムに変換する．有向グラフの各ノードは関数へと変換され，エッジの向きに関数呼び出しを行う．図 3 を変換すると次のようになる．

```
def receiveFromSink(msg) do
  case msg.name do
    "Y" -> __map2(msg.element, msg.rc)
  end
end
def __map2(value, rc) do
  x = f1(value)
  __filter1(x, rc)
end
def __filter1(value, rc) do
  y = f2(value)
  if y do
    sendToSink_X(value, rc)
  end
end
def sendToSink_X(value, rc) do
  send({name: "X", element: value, rc: rc},
       sinkId)
  sendToDependencyRegion()
end
```

receiveFromSink関数でシンクから受け取ったメッセージを名前で振り分ける．mapや filterなどは単純な関数へと変換され，最後に値に名前をつけてシンクに送信する．iterの場合は他ノードに値を送信すること無く関数呼び出しが終わる．

VNC による協調を行うため，各ノードはシンクから最初にメッセージを受信するタイミングで依存領域内のノードを含む木構造のネットワークを構成す

る．Sinkからメッセージを受信した際は，メッセージに付随する RC 値と依存領域内の VNC 値を比較して，同期が取れたら receiveFromSinkにメッセージを送信する．Sinkに値を送信する際は，Sinkから送信された rc をそのまま返し，更に依存領域内のノードに自身の保持する VNC 値を送信する．

5.3 シンク向けプログラムへの変換

シンクプログラム上ではストリームを集合として扱う．ブロック間参照表を見て他のブロックが参照しているストリーム変数をシンクが保持していたら，そのストリーム変数に対応する集合を各ノードに向けて送信する．逆に他のブロックの変数を参照していた場合，シンクが送信した命令に対するレスポンスとなる．receive関数が常に受け取ったレスポンスを対応する集合に追加しており，対応する命令を他ノードに送信した後に receive関数による集合の生成を一定時間待つ．1 周期が終わったらループし，再度sensorsに値を送る．

以下はプログラム 1 の Sinkブロックをコンパイルした例である．

```
def receive(Message res) do
  case res.name do
    "FIRE_REGION" ->
      put(fireRegion, res.element)
  end
end
def mainloop() do
   send_1(sensors)
   wait()
   actors = getActors(fireRegion)
   send_1(actors)
   mainloop()
end
```

send_1で値を送信する際は，シンクが保持する RC の 1 番目の要素をインクリメントして，値を送信する際に同時に RC 値も送信する．イベント領域が複数ある場合は send_2, send_3といったように対応する番号が変わる．レスポンスにおいても，付随している RC 値が現在シンクが保持している RC 値と同じ値の場合のみ receive関数が呼ばれる．

6 関連研究

無線センサーアクチュエータネットワーク向けのマクロプログラミング言語には SOSNA [3] が挙げられる．ただしアクチュエータはアクターの別名である．SOSNA はノードのグループのストリームを操作することでネットワークの動作を記述するような関数型マクロプログラミング言語である．Brigade のようにセンサーの情報をシンクに送信してからアクターにデータを送信するのではなく，アクターが周囲のセンサーの値を直接集約したり，アクター同士で通信して協調を行う．ノード間通信を同期することによって WSAN プログラムの 1 周期の計算にかかる時間を見積もることができる．SOSNA は同期的に計算を行うため，通信待機時間を長くすることでパケットの損失を抑え，ハザードを回避することができるが，結果として周期ごとの計算に時間がかかり，リアルタイム性が損なわれてしまう．

また WSAN 向けマクロプログラミングミドルウェアとして PICO-MP [2] が挙げられる．ノードの種類を Publisher と Subscriber に分け，Publisher は型付けされたイベントの通知を PICO-MP 基盤に定期的に送信する．Subscriber は一階述語論理の形式で表される述語によって自身が関心のあるイベントの種類を記述し，PICO-MP 基盤から送信されるイベント通知が述語を満たすかどうか検査する．述語を満たすイベントを受け取ったら Subscriber は Publisher に対してイベントの情報を要求する．ネットワークは木構造で構成され，イベント通知を仲介するノードは子ノードの Subscriber の部分式を用いることでイベント通知をフィルタリングし，通信量を削減している．一方，データ通信の順序に関しては定められていないため，依存する命令の到達順序が入れ替わり，結果としてハザードが発生する可能性がある．

7 結論と今後の課題

本研究では，WSAN 向けマクロプログラミング言語 Brigade と，その実装手法を提案した．Brigade 言語を用いることで WSAN プログラムを簡潔に記述で

きることを示した．またプログラム中のノード間通信を行う箇所に自動で VNC による協調機構を組み込むことで，WSAN のノード間協調の問題の 1 つであるハザードをプログラマが意識すること無く回避できるようになった．

Brigade ではストリームの値を `filter`することによってシンクに送信するデータ量を減らすことができるが，距離の長さや再送によって伝送が遅延し，データが到達する前にシンクが計算を進めてしまう可能性がある．計算に間に合わなかったデータは捨てられるため，シンクからの距離が遠いノードからのデータが頻繁に捨てられてしまう可能性がある．シンクが全ノードからの実行完了通知を受け取るようにする場合，シンクに送信するデータ量を減らすことができないため，各センサーのデータを一度アクターに集約するなど，全体の通信量を減らすような仕組みを実装すべきである．

また Brigade は `getActors`によるアクターへのデータを送信方法を提供しているが，センサーが計測した値をアクターに送信する手段がない．1 つのアクターが複数種類のセンサーの観測範囲にある場合，最も近いセンサーの値をアクターに割り当てたり，線形補間するなどしてアクターに送信する値を決定する必要がある．

謝辞 本研究は JSPS 科研費 18K11236 の助成を受けている．

参 考 文 献

[1] Akyildiz, I. F. and Kasimoglu, I. H.: Wireless sensor and actor networks: research challenges, *Ad Hoc Networks*, Vol. 2, No. 4(2004), pp. 351–367.

[2] Dulay, N., Micheletti, M., Mostarda, L., and Piermarteri, A.: PICO-MP: De-centralised Macro-Programming for Wireless Sensor and Actuator Networks, *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, 2018, pp. 289–296.

[3] Karpiński, M. and Cahill, V.: Stream-Based Macro-Programming of Wireless Sensor, Actuator Network Applications with SOSNA, *Proceedings of the 5th Workshop on Data Management for Sensor Networks*, DMSN '08, New York, NY, USA, Association for Computing Machinery, 2008, pp. 49–55.

[4] Newton, R., Arvind, and Welsh, M.: Building up to macroprogramming: an intermediate language for sensor networks, *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, 2005, pp. 37–44.

[5] Newton, R., Morrisett, G., and Welsh, M.: The Regiment Macroprogramming System, *6th International Conference on Information Processing in Sensor Networks (IPSN '07)*, ACM, Apr. 2007, pp. 489–498.

[6] Vedantham, R., Zhuang, Z., and Sivakumar, R.: Hazard avoidance in wireless sensor and actor networks, *Computer Communications*, Vol. 29, No. 13(2006), pp. 2578–2598.