SLEDS: A DSL for Data-Centric Storage on Wireless Sensor Networks[⋆]

Marcos Aurélio Carrero[1], Martin A. Musicante[2], Aldri Luiz dos Santos[1], and Carmem S. Hara[1]

[1] Universidade Federal do Paraná, Brazil
{macarrero,aldri,carmem}@inf.ufpr.br
[2] Universidade Federal do Rio Grande do Norte, Brazil
mam@dimap.ufrn.br

Abstract. The dynamicity requirements of urban sensor networks rise new challenges to the development of data management and storage models. Software component techniques allow developers to build a software system from reusable, existing components sharing a common interface. Moreover, the development of urban sensor networks applications would greatly benefit from the existence of a dedicated programming environment. This paper proposes SLEDS, a Domain-Specific Language for Data-Centric Storage on Wireless Sensor Networks. The language includes high-level composition primitives, to promote a flexible coordination execution flow and interaction between components. We present the language specification as well as a case study of data storage coordination on sensor networks. The current specification of the language generates code for the NS2 simulation environment. The case study shows that the language implements a flexible model, which is general enough to be used on a wide variety of sensor network applications.

Keywords: WSN Storage · Software Components · Domain-Specific Languages.

1 Introduction

Wireless sensor networks (WSNs) are essential components of urban computing. They can be applied in a variety of contexts. For traffic monitoring, they can be used to monitor the flow of vehicles in order to control the traffic lights and minimize jams. For environment monitoring, they can be used to collect the pollution level in order to detect critical areas and take actions that minimize its effect on the population.

Sensor networks deployed on urban areas are usually dense. They are composed of thousands of devices that communicate via radio, and have limited resources for processing and storing data. There are three categories of data storage models for WSNs [13, 22]: local, external, and data-centric. The local and

[⋆] This research was partially funded by INES 2.0, CNPq grant 465614/2014-0 and Fundação Araucária.

external categories store sensored data on the sensor device, and on an external device with more resources (usually called the *base station*), respectively. These categories are not appropriate for dense networks. This is because local storage requires all devices to be contacted in order to collect data to answer queries, which may result in poor response times. On the other hand, external storage requires sensors to periodically report their readings to the base station, which may cause unnecessary high traffic of messages. The data-centric approach, on the other hand, combines both approaches, by electing a subset of sensors to act as representatives of sets of devices, which store their readings. In this model, groups of sensors compose *clusters*, represented by *cluster heads - CHs*. Thus, in order to answer queries, only CHs have to be contacted, providing scalability for dense networks, such as urban WSNs. There are a number of data-centric storage models proposed for urban scenarios [11].

Validation of the proposed models usually involves programming them in a simulation environment, given the costs and difficulties of deploying such large networks in real settings. NS2, NS3, and OMNeT++ are among the simulation environments used for WSNs. While developing previous works on urban data-centric models we have noticed that: *(1)* the majority of programs are developed from scratch, and there is little to no support for code reusability; and *(2)* there are similarities among models on the flow of activities, which can be modeled as state machines. We have tackled these problems by proposing a component-based model for WSNs called RCBM [7], and a state machine to formalize the interaction among components [5]. Although the state machine helps the *specification* of the overall flow of activities, the programmer is still responsible for developing the code specified by the state machine. In this paper, we propose a language that closely resembles a state machine, which allows the programmer to define the flow of activities in a higher-level of abstraction. Our current specification generates code for the NS2 simulator. However, we envision that in the future the same program can be used to generate code for sensor devices using a platform-independent library such as wiselib [4].

The idea of specifying the control flow in a higher-level language can minimize the complexity of developing event-based programs. Event-based programming is often used as an abstraction mechanism for devices with limited resources. In WSNs, this programming model is adopted by operating systems such as TinyOS and Contiki, as well as for simulation environments: NS2, NS3 and OMNeT++. However, the flow of events in these programs is hard to understand and maintain [14]. Examining NS2 programs coded by different developers, we have noticed that they found it difficult to control the flow of activities when they were not triggered by an event, but by a logical condition or a timer. Each programmer used a different approach for handling this type of state change, generating completely different programs, which are thus hard to maintain.

Our proposed language, called SLEDS (State Machine-based Language for Event-Driven Systems), overcomes this problem, by directly defining states and transitions among them. Transitions may be event-based and logic-based. SLEDS also supports primitives for point-to-point and broadcast communication among sensors. The language focus on the coordination flow of data-centric entities that are associated with a set of components that implement common functionalities. In this paper, we present the language specification, as well as a syntax-based translation of SLEDS to NS2. A case study that implements data-centric storage models for WSNs shows that the language is general enough to be used on a wide variety of applications.

The remainder of the paper is organized as follows. Section 2 discusses the related work. Section 3 presents SLEDS as well as a syntax-based translation. Section 4 details a case study of data storage coordination on sensor networks. We conclude in Section 5 highlighting future works.

2 Related Work

Data-centric storage has attracted a lot of attention, given that its decentralized approach is more scalable for large-scale urban WSNs than external and local storage models. In this model, some sensors in the network are responsible for storing the readings of a group of sensors. MKSP [9] follows this approach by mapping raw data to storage nodes. In order to exploit the spatial data similarity of sensor readings, AQPM [6] and SILENCE [17] consider some sensors elected as cluster heads to be the group representative, minimizing the communication overhead. Although recent efforts have been made to build efficient data storage systems, the specific nature of WSNs and the lack of a common general purpose development framework make the design of these applications a hard task. RCBM [7] promotes software reuse from existing components to improve the efficiency of system development and evaluation. The separation of the coordinator from the application components proposed by RCBM allows developers to explore similarities among models on the flow of activities.

Domain-Specific Languages (DSL) are programming languages to be used in a well-defined context. As opposed to general-purpose programming languages, DSLs are devised to closely follow practices of their application domain [10]. DSLs are commonly used in the context of Wireless Sensor Networks [8], as well as for the definition of state-transition systems. In the context of WSNs, Hood [23] provides a neighborhood programming abstraction. Algorithms are designed based on a set of criteria for choosing neighbors and definition of variables to share among them. Hood is aimed at simplifying the use of operations such as synchronization and communication with neighboring sensors. SenNet [21] abstracts WSN programming complexity to develop node and group-level applications. Although the purpose of Hood, SenNet and SLEDS is similar, to provide a high-level abstraction for developing sensor-based application, the approaches adopted by each of them differ. SLEDS is based on a state machine while Hood is based on the concept of neighborhood. SenNet does not adopt a flexible execution model such as the one proposed by SLEDS.

Also in the Data-Centric context, Regiment [20] is a DSL which provides a geo-temporal view of the WSN. The language provides primitives to manipulate sets of geo-localized data streams. This centralized view is translated by the compiler into specific code to be run by each sensor in the network. The use of DSLs for the definition of state machines is a well-studied topic [15]. For instance, in [19], the authors propose a DSL to implement a specific type of state machines to describe complex systems. However, it does not target sensor applications as SLEDS. We observed that the development of programs to control data-centric coordination and storage on WSNs follow patterns that can be modeled by state-transition machines. Moreover, developing such applications is usually considered complex by an average programmer. In order to tackle these problems, we defined SLEDS, a DSL for managing data-centric storage on WSNs.

3 SLEDS: The Language

This section presents SLEDS (State Machine-based Language for Event Driven Systems), a DSL used to implement state machines for data storage coordination on sensor networks. Section 3.1 presents the language syntax, followed by a specification of its translation to NS2 network simulation code in Section 3.2.

3.1 Syntax

The grammar illustrated in Figure 1 describes the SLEDS syntax. In our programming model, each sensor executes an instance of a SLEDS program. A state machine communicates with each other through asynchronous message passing. A SLEDS program consists of (*i*) a `Program` declaration, its identifier and a sequence of input parameters, followed by (*ii*) a sequence of constants `const`, (*iii*) a sequence of variables, and (*iv*) a sequence of state definitions `StateDef`.

A `StateDef` is composed of an identifier, a sequence of input parameters followed by a list of actions `ActionList`. The actions correspond to sensor activities triggered in response to an event or based on a logical condition. An `ActionList` is a sequence of standard control flows, such as sequential, conditional, and iteration, as well as primitives for sending and receiving messages, as detailed below:

– Action ::= **nextState** *State*: describes a state change to a new *State*. Each *State* declaration has a name `Id` with arguments representing the input parameters of the state. The ***exit*** state finishes the program.
– Action ::= **broadcast (***Exp*, *Exp*, *ExpList***)**: corresponds to the asynchronous communication sent from a sensor to all its neighbor sensors, that is, the ones within its communication range. The arguments are the message type, message identifier and a list of parameters.
– Action ::= **send (***Exp*, *Exp*, *ExpList*, *ExpList***)**: corresponds to the asynchronous communication sent from a sensor to a set of destination sensors. The arguments are the message type, message identifier, the set of destinations and a list of parameters.
– Action ::= **on recvBroadcast (***Id*, *Id*, *IdList***){** *ActionList* **}**: corresponds to the receipt of a *broadcast* message.

Program	::=	**Program** *Id*(*Type Id*(, *Type Id*)*) **{**
		(**const** *Id* = *(Num-Literal* | *Str-Literal)*;)*
		(*Type VarList*;)* *StateDef** **}**
VarList	::=	*Var* (, *Var*)*;
Var	::=	*Assignment* | *Id*
StateDef	::=	**State** *Id*(*Type Id*(,*Type Id*)*) **{** *ActionList* **}**
State	::=	*Id* (*ExpList*?) | ***exit***
ActionList	::=	*Action* (*Action*)*
Action	::=	**nextState** *State*;
		| **broadcast**(*Exp*, *Exp*, *ExpList*);
		| **send**(*Exp*, *Exp*, *ExpList*, *ExpList*);
		| **on recvBroadcast**(*Id*, *Id*, *IdList*) **{***ActionList***}**
		| **on recv**(*Id*, *Id*, *IdList*, *IdList*) **{** *ActionList* **}**
		| **during**(*Exp*) **on recvBroadcast**(*Id*, *Id*, *IdList*)
		{ *ActionList* **}** **nextState** *State*;
		| **during** (*Exp*) **on recv**(*Id*, *Id*, *IdList*, *IdList*)
		{ *ActionList* **}** **nextState** *State*;
		| **while** (*Exp*) **{** *ActionList* **}**
		| **for** *Id* **in** *Exp* **{** *ActionList* **}**
		| **if** (*Exp*) **{** *ActionList* **}** (**else** **{** *ActionList* **}**)?
		| *Assignment*;
		| *Method-call*;
Method-call	::=	*Exp*
Assignment	::=	*Id* = *Exp*
Exp	::=	*Exp* −> *Exp* | *Exp* **.** *Id* | *Exp*(*Exp*?) | *Id*
ExpList	::=	*Exp* (, *Exp*)*
IdList	::=	*Id* (, *Id*)*

Fig. 1. SLEDS Syntax

– Action ::= **on recv** (*Id*, *Id*, *IdList*, *IdList*)**{** *ActionList* **}**: corresponds to the receipt of a *send* message.
– Action ::= **during** (*Exp*) **on recvBroadcast** (*Id*, *Id*, *IdList*) **{** *ActionList* **}** **nextState** *State*: corresponds to the receipt of a *broadcast* message during a time interval, and change to a new *State* at the end of this period.
– Action ::= **during** (*Exp*) **on recv** (*Id*, *Id*, *IdList*, *IdList*) **{** *ActionList* **}** **nextState** *State*: corresponds to the receipt of a *send* message during a time interval, and change to a new *State* at the end of this period.

As an example, consider the state machine for discovering neighbors illustrated in Figure 2. Note that there are two types of transitions:

– event state change: specifies a transition to a new state when the sensor receives a message or upon the timer expiration (represented in blue lines).
– logic state change: a machine transitions to a new state triggered by the result of a computation, and represented in red lines.

Some event-based languages, such as NS2 simulator, provide limited abstraction to implement state machine models. The SLEDS language facilitates this

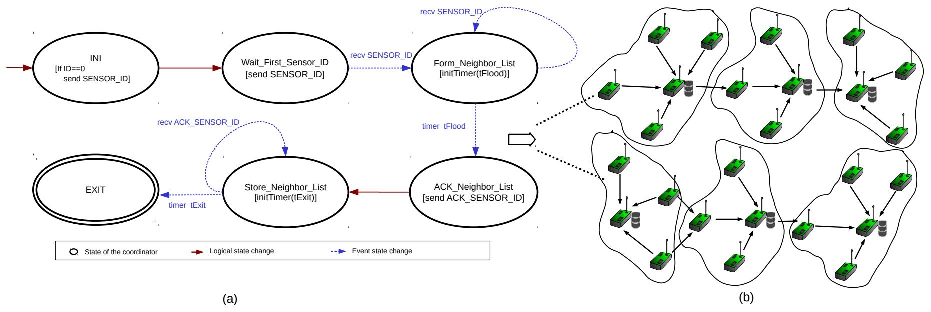


Fig. 2. State machine of a discovery neighbors algorithm.

task allowing developers to describe the state machine in a high level abstraction and translates this representation into an NS2 simulation code. Listing 1.1 illustrates the SLEDS program that implements the neighbor discovery machine.

```
use compSensor as ComponentsSensor;
use compLibMSG as ComponentsLibMessage;

Program Coordinator() {
  const tFlood=25;
  const tExit=0.1;
  int myID = compSensor->getSensorId();
  list<int> listSensorAnnouncements;
  int msgID;

  STATE INI() {
    if (myID == 0) {
      msgID = compLibMSG->GetNextMsgId();
      broadcast(SENSOR_ID, msgID, myID);
      compLibMSG->addSeenMsg(SENSOR_ID, msgID); }
    nextState Wait_First_Sensor_ID(); }

  STATE Wait_First_Sensor_ID() {
    on recvBroadcast(SENSOR_ID, msgID, ID) {
        listSensorAnnouncements.insert(ID);
        if (!compLibMSG->seenMsg(SENSOR_ID, msgID)) {
            compLibMSG->addSeenMsg(SENSOR_ID, msgID);
            broadcast(SENSOR_ID, msgID, myID); }
        nextState Form_Neighbor_List();  } }

  STATE Form_Neighbor_List() {
    during (tFlood) on recvBroadcast(SENSOR_ID, msgID, ID) {
      listSensorAnnouncements.insert(ID); }
    nextState ACK_Neighbor_List(); }

  STATE ACK_Neighbor_List() {
    send(ACK_SENSOR_ID, compLibMSG->GetNextMsgId(),
         listSensorAnnouncements, myID);
    nextState Store_Neighbor_List(); }

  STATE Store_Neighbor_List() {
    during (tExit) on recv(ACK_SENSOR_ID, msgID,
                          listSensorAnnouncements, fromID) {
      for v in listSensorAnnouncements
        if (v == myID)
          compSensor->listKnownNeighbors.insert(fromID); }
    nextState exit;}
```

Listing 1.1. Neighbor discovery SLEDS program

The program assumes the existence of components: *compSensor* and *compLibMSG*. The first provides basic sensor functionality, such as returning the sensor identification (function *getSensorId*), and storage of its list of neighbors (*listKnownNeighbors*). The *compLibMSG* provides functionality related to messages exchanged among sensors. There are functions to create a new message identification (*GetNextMsgId*) and to store, for each sensor, the type and identification of messages already received (*addSeenMsg*). In fact, in a component-based programming environment, a SLEDS program plays the role of a coordinator, which is responsible for the flow of activities that glue the software components, specifying the interactions among them. After including references to the components *(l.1-2)*, the program declares a set of constants and variables. Constant *tFlood* defines the delay of message transmissions in the network, and constant *tExit* determines the delay needed between sending a message and receiving an acknowledgement in order to avoid collisions. Variable *myID* keeps the sensor unique identifier, which is obtained executing function *getSensorId()* provided by the *compSensor* component *(l.7)*.

Every sensor executes the same SLEDS program, starting in the `INI` state *(l.11)*. In this state, the sensor with *myID* zero, obtains a new message identifier and sends a message of type `SENSOR_ID` to all its neighbors *(l.12-14)*, and stores the message identifier type and identifier in order to avoid sending duplicated messages *(l.15)*. The sensor with *myID* zero and the remaining sensors perform a logical state change to `Wait_First_Sensor_ID` *(l.16)*. The sensors that receive the first message `SENSOR_ID` store the *ID* contained in the message and send their identifier to their neighbors *(l.19-23)* and perform an event state transition to `Form_Neighbor_List` *(l.24)*. In state `Form_Neighbor_List`, sensors continue to store the `ID` from their neighbors during a time interval *tFlood (l.26-28)*. When the timer expires, nodes perform an event state change to `ACK_Neighbor_List` *(l.29)* and send an `ACK` message to the known neighbors recorded in variable *listSensorAnnouncements (l.32)*. After sending the `ACK` message, the sensors make a logical state change to `Store_Neighbor_List` *(l.34)*. In state `Store_Neighbor_List`, during a time interval *tExit (l.36-38)*, sensors that receive the message check if they are the final destination and update its list of neighbors *(l.40-41)*. At the end of the flooding, each sensor has in its local *listKnowNeighbors* variable, its list of neighbors. Next section presents a proposal to translate SLEDS programs to NS2 simulation codes.

3.2 Translation to NS2

In a NS2 program, the coordination of the sensor activities is implemented in two main functions: `recv` and `TimerHandle`. Function `recv` is responsible for managing messages and contains the code for state transitions triggered upon a message receipt. Function `TimerHandle` is activated by the expiration of a timer. Observe that in the state machine illustrated in Figure 2, both are represented as event-based transitions, and there is no distinction among them in the SLEDS program in Listing 1.1. However, in the NS2 program, they have to be coded in different functions, which adds complexity to understand the flow of activities. Moreover, states may generate code not only for the `recv` or `TimerHandle` functions, but for both. Examples for each case are presented next.

```
1  STATE INI() {
2    if (myID == 0) {
3       msgID = compLibMSG→GetNextMsgId();
4       broadcast(SENSOR_ID, msgID, myID);
5       compLibMSG→addSeenMsg(SENSOR_ID, msgID);
6    }
7    nextState Wait_First_Sensor_ID();
8  }
```

```
1  void WSN_ComponentsAgent::TimerHandle(State st) {
2    switch (st) {
3       case INI: {
4         if (myID == 0) {
5            msgID = compLibMSG→GetNextMsgId();
6            broadcast(SENSOR_ID, msgID, myID);
7            compLibMSG→addSeenMsg(SENSOR_ID, msgID);
8          }
9          nextState=Wait_First_Sensor_ID;
10      }
11   }
12 }
```

Fig. 3. SLEDS code and NS2 translation of INI state

The `INI` state of Listing 1.1 is an example that generates code only for the `TimerHandle` function as shown in Figure 3. The translated code is composed of a *switch* command, with *case* clauses, one for each state identifier. The *case* for the `INI` state contains the same code provided by the SLEDS program, which ends with a state transition to state `Wait_First_Sensor_ID` *(l.9)*. The translation of this state generates code only for the `recv` function, as illustrated in Figure 4.

```
1  STATE Wait_First_Sensor_ID() {
2   on recvBroadcast(SENSOR_ID, msgID, ID) {
3    listSensorAnnouncements.insert(ID);
4    if  (!compLibMSG→seenMsg(SENSOR_ID, msgID)) {
5      compLibMSG→addSeenMsg(SENSOR_ID, msgID);
6      broadcast(SENSOR_ID, msgID, myID);
7    }
8    nextState Form_Neighbor_List();
9   }
10 }
```

```
1  void WSN_ComponentsAgent::recv(Packet* pkt, Handler *) {
2    WSN_Components_Message p = pkt;
3    switch(nextState) {
4      case (Wait_First_Sensor_ID): {
5        if (param.getMsgType() == SENSOR_ID) {
6          listSensorAnnouncements.insert(param.getSensorID());
7          if  (!compLibMSG→seenMsg(p.getSensorID(), p.getMsgID())) {
8              compLibMSG→addSeenMsg(p.getSensorID(), p.getMsgID());
9              broadcast(SENSOR_ID, msgID, myID);
10          }
11          nextState=Form_Neighbor_List;
12       }
13     }
14  }
15 }
```

Fig. 4. SLEDS code and NS2 translation of Wait_First_Sensor_ID state

Similar to `TimerHandle`, the `recv` function in NS2 is also composed of a *switch* command, with *case* clauses, one for each state. Part of the generated code in NS2 is to obtain the parameters from the the packet received, but most of the code inside each *case* clause is identical to the SLEDS program. The translation to NS2 is not so direct when it involves both functions `TimerHandle` and `recv`, as shown in Figure 5, which corresponds to state `Form_Neighbor_List`. The `recv` function sets a timer `tFlood` *(l.7)*. During this period the sensor stores the neighbor announcements *(l.10)* at every `SENSOR_ID` message received. However, the transition to the next state `ACK_Neighbor_List` cannot be made in this function, since the sensor may receive multiple message of this type. Thus, the transition is coded in the `TimerHandle` function, which is triggered at the expiration of `tFlood` timer.

```
STATE Form_Neighbor_List() {
  During (tFlood) on recvBroadcast(SENSOR_ID, msgID, ID) {
      listSensorAnnouncements.insert(ID);
  }
  nextState ACK_Neighbor_List();
}
```

```
void WSN_ComponentsAgent::recv(Packet* pkt, Handler *) {
   WSN_Components_Message p = pkt;
    switch(nextState) {
        case (Form_Neighbor_List): {
               if (param.getMsgType() == SENSOR_ID) {
                   if (nextState != previousState) {
                        libTimer.resetTimer(tFlood);
                        previousState=Form_Neighbor_List;
                   }
                   listSensorAnnouncements.insert(p.getSensorID());
               }
}
```

```
void WSN_ComponentsAgent::TimerHandle(State st) {
  switch (st) {
        case Form_Neighbor_List: {
               nextState=ACK_Neighbor_List;
               libTimer.resetTimer(0);
        }
  }
}
```

Fig. 5. SLEDS code and NS2 translation of Form_Neighbor_List state

Due to space limit, we have not included the translation for states `ACK_Neighbor_List`, which generates code in function `TimerHandle`, and `Store_Neighbor_List`, which generates code in both functions.

We have adopted a syntax directed translation to generate the NS2 code from a SLEDS program. In this technique, *attributes* and *semantic rules* are associated to each production of the grammar [1]. The general approach consists of building a derivation tree and then determine the attribute values in each node of the tree during its traversal. Semantic rules express the relationship between the computation of attribute values and the productions, by associating code fragments to each attribute. The set of attributes and semantic rules is denoted as an *attribute grammar* [18]. There are two types of attributes: synthesized and inherited. In our approach, synthesized attributes are used to pass semantic information up the parse tree, while inherited attributes help pass semantic information down.

In our grammar, there are three attributes: `rc`, `tc`, and `dest`. The first two are synthesized attributes and determine whether the code fragment is going to be included in function `recv` or `TimerHandle`, respectively. Attribute `dest` is inherited, and may contain either the value `rc` or `tc`, and it is used to pass down the tree the function in which the code will be generated, based on the node context.

The attribute grammar in Figure 6, describes the attributes and grammar rules to generate code for the state `Form_Neighbor_List`, and Figure 7 the resulting parse tree. Observe at the bottom of the tree that the inherited attribute `dest` of node `Action` contains the value `rc` in order to pass the information down the tree that the `Method-call` should generate code for attribute `rc`. This attribute, will then receive the code fragment, which is passed up the tree in order to compose the final `rc` value at the tree root. This is the code that will be included in the `recv` function. The same process is used to compose the final value of the `tc` attribute, with the code to be included in the `TimerHandle` function.

Examples presented in this section show how a SLEDS program can generate NS2 code using an attribute grammar specification. The next section presents a case study that shows how SLEDS programs can be used to generate data-centric storage models for WSNs in a component-based development framework.

(r1)	$State ::= \textbf{State}_\mathbf{1}\ Id_1$ "(" $Type\ Id_2(,Type\ Id_3)^*$) "{" $ActionList$ "}"	

		`if (`$ActionList.tc$` != null)`
		    $State.tc$ `= ''switch (nextState):{ case'' +` $Id_1.txt$ `+ '':''`
		        `+ ''{''` $ActionList.tc$ `+ ''}}''`
		`if (`$ActionList.rc$` != null)`
		    $State.rc$ `= ''switch (nextState):{ case'' +` $Id_1.txt$ `+ '':''`
		        `+ ''{''+` $ActionList.rc$ `+ ''}}''`

(r2)	$Action ::=$ **during** "(" Exp ")" **on recvBroadcast** "(" $Id_1,\ Id_2,\ IdList$ ")" "{" $ActionList$ "}" $nextState\ State$ ")"	
		$Action.rc$ `= ''if (nextState != previousState) {''` $Action.rc$ `+= ''libTimer.resetTimer(''+`$Exp.txt$`+'')''` $Action.rc$ `+= ''previousState=''+`$Action.pst$ $Action.rc$ `+=` $ActionList.rc$ `+ ''}''`
(r3)	$Method - call ::= \ Exp$	

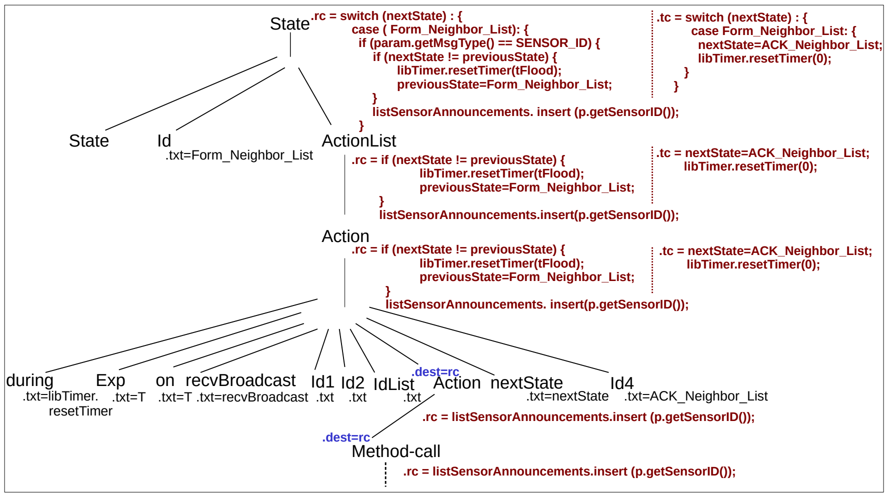
		`if(`$Method - call.dest$`==rc) {`$Method - call.rc$ `=` $Exp.rc$`}`
		`else {` $Method - call.tc$ `=` $Exp.tc$ `}`

Fig. 6. Attribute grammar to generate code for state Form_Neighbor_List

4 Validation

RCBM [7] is a component-based framework to develop WSNs storage models that promotes code reusability. It is depicted in the central box of Figure 8. RCBM addresses data-centric entities that share concepts and functionalities, which represent various instances of WSN storage systems. These shared functionalities are the components of the system. Although it has been shown that the framework is efficient for promoting code reusability, the application developer is still responsible for coding the coordination among the components. SLEDS, with its high-level composition primitives, can be used to generate code for the RCBM coordinator, as we will show in this section.

RCBM has been implemented on the NS2 simulator and considers three types of components: library components, application components, and the coordinator. Library components provide a toolbox, that can be used to implement application components, associated with WSN entities. For data-centric models, these entities include: *(i)* sensor devices; *(ii)* cluster members (CM), which consist of a set of sensors; and *(iii)* cluster-heads (CHs) that are sensors responsible for storing the information of all cluster members. These entities define a hierarchical storage model, where each cluster designates a sensor as cluster-head for storing the readings of its group members. The coordinator is responsible for the execution flow and message exchanges. Next section shows a case study that implements data-centric storage models for WSNs.



```
.rc = switch (nextState) : {
        case ( Form_Neighbor_List): {
         if (param.getMsgType() == SENSOR_ID) {
            if (nextState != previousState) {
                libTimer.resetTimer(tFlood);
                previousState=Form_Neighbor_List;
            }
            listSensorAnnouncements. insert (p.getSensorID());
         }
```

```
.tc = switch (nextState) : {
        case Form_Neighbor_List: {
          nextState=ACK_Neighbor_List;
          libTimer.resetTimer(0);
        }
      }
```

```
.rc = if (nextState != previousState) {
            libTimer.resetTimer(tFlood);
            previousState=Form_Neighbor_List;
     }
     listSensorAnnouncements.insert(p.getSensorID());
```

```
.tc = nextState=ACK_Neighbor_List;
    libTimer.resetTimer(0);
```

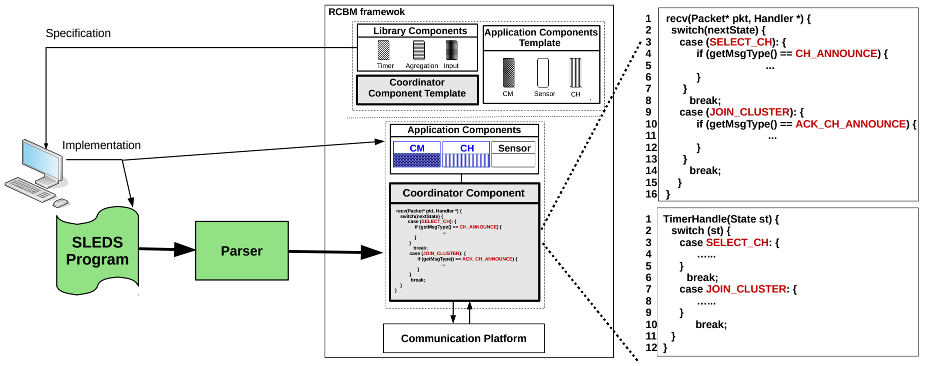
```
.rc = if (nextState != previousState) {
            libTimer.resetTimer(tFlood);
            previousState=Form_Neighbor_List;
      }
      listSensorAnnouncements. insert(p.getSensorID());
```

```
.tc = nextState=ACK_Neighbor_List;
    libTimer.resetTimer(0);
```

```
.rc = listSensorAnnouncements.insert (p.getSensorID());
```

```
.rc = listSensorAnnouncements.insert (p.getSensorID());
```

Fig. 7. Parse tree for the Form_Neighbor_List state



```
recv(Packet* pkt, Handler *) {
  switch(nextState) {
    case (SELECT_CH): {
        if (getMsgType() == CH_ANNOUNCE) {
                        ...
        }
     }
       break;
    case (JOIN_CLUSTER): {
        if (getMsgType() == ACK_CH_ANNOUNCE) {
                        ...
        }
     }
       break;
   }
}
```

```
TimerHandle(State st) {
  switch (st) {
    case SELECT_CH: {
        ......
    }
      break;
    case JOIN_CLUSTER: {
        ......
    }
        break;
  }
}
```

Fig. 8. SLEDS back-end architecture.

4.1 LEACH Coordinator Component Implementation

LEACH (Low-Energy Adaptive Clustering Hierarchy) [16] is a probabilistic model that forms one-hop clusters. LEACH assumes that all nodes are within the communication range of each other. Sensors elect themselves as cluster-heads with a probability p. In RCBM, the `compCH` component defines the function `selectCH(map<K, V>)` that the developer should implement according to the target model. For the coordination that implements the cluster formation of LEACH, every sensor s_i executes `selectCH(`s_i`,` p`)`, where $K = s_i$ and $V = p$. Listing 1.2 depicts the SLEDS coordination code of the CH election phase.

```
// Program executed by each sensor
use compSensor as WSN_ComponentsSensor;
use compCH as  WSN_ComponentsCH;
```

```
4   use compCM as  WSN_ComponentsCM;
5   use compLibMSG as ComponentsLibMessage;
6
7   Program Coordinator() {
8     const p=0.2;
9     const tCluster=25;
10    const tExit=0.1;
11
12    double RSS;
13    int myCH;
14    int myID=compSensor->getSensorId();
15    list<int, double> knownCHs;
16    list<int> sensorList;
17
18    STATE Select_CH() {
19    if (compCH->selectCH(myID, p)) {
20      broadcast(CH_ANNOUNCE, compLibMSG->GetNextMsgId(), myID);
21      compSensor->role = CH; }
22    else {
23      compSensor->role = CM; }
24    nextState Join_Cluster(); }
```

Listing 1.2. The CH Election Coordination

The `Select_CH` state (*l.18*) describes the actions that should be executed during the election phase. First, each node calls `selectCH()` (*l.19*). If the function returns `true` then the sensor broadcasts its role as cluster-head (CH) to the network (*l.20-21*) and performs a state transition to `Join_Cluster` (*l.24*). Otherwise, the sensor role is set as a cluster member (CM) (*l.22-23*). Listing 1.3 illustrates the `Join_Cluster` state code.

```
25    STATE Join_Cluster() {
26    During (tCluster) on recvBroadcast(CH_ANNOUNCE, msgID, ID) {
27      RSS = compSensor->getRSS(ID);
28      knownCHs.insert(ID, RSS); }
29    nextState Cluster_Formation(); }
```

Listing 1.3. The Join Cluster Coordination

In the next state (*l.25*), sensor nodes wait for `tCluster` time units for CH announcements and update the value of `knownCHs` based on the received signal strength (RSS) (*l.26-28*). When the timer expires, remaining sensors join the cluster, as illustrated in Listing 1.4.

```
30  STATE Cluster_Formation() {
31    if (compSensor->role = CM) {
32        myCH = compCM->joinCluster(knownCHs);
33        sensorList.insert( myCH );
34        send(ACK_CH_ANNOUNCE, compLibMSG->GetNextMsgId(), sensorList, myID);
35        nextState EXIT;
36    } else {
37    nextState Store_Members(); }
```

Listing 1.4. The Cluster Formation Coordination

In LEACH, a cluster member (*l.31*) decides to join the cluster that requires the lowest energy consumption to communicate. Thus, it sets as CH the sensor with the maximum RSS recorded by `knownCHs` (*l.32*). Then, it sends an `ACK_CH_ANNOUNCE` message to the chosen one and move to the `EXIT` state (*l.33-35*). Otherwise, the CHs perform a logical state change to `Store_Members` (*l.36-37*). Listing 1.5 depicts the SLEDS code implementation.

```
38   STATE Store_Members() {
39    During (tExit) on recv(ACK_CH_ANNOUNCE, msgID, destListID, fromID) {
40      for v in destListID{
41       if (v = myID) {
42        compCH->members.insert( fromID ); } } }
43     nextState EXIT ;}
```

Listing 1.5. The Store Members Coordination

The CHs execute the actions corresponding to the `Store_Members` state (*l.38*). First, CHs wait for timer `tExit` units to receive `ACK_CH_ANNOUNCE` messages from group members (*l.39*). If it receives a message, the sensor updates its list structure of members (*l.40-42*). When the timer `tExit` expires, the program terminates its execution (*l.43*). Figure 9 illustrates a state machine of the LEACH coordination flow, an instance of a data-centric WSN storage system.

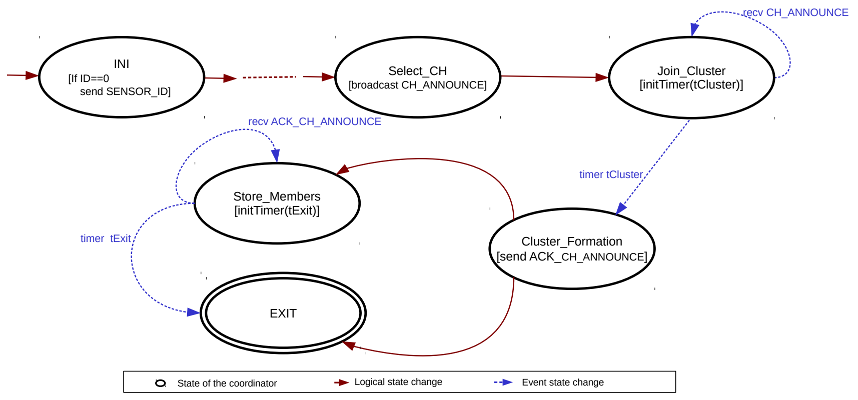


Fig. 9. State machine of a data-centric storage model.

The flow of execution depicted in Figure 9 is similar to the one adopted by the component-based framework CBCWSN [2], which has been shown to express a number of data-centric storage instances. As we will show next, the same can be said of the SLEDS program. In order to implement LCA [3], only three lines of code have to be modified, mainly to take into consideration a distinct criterion for CH election. LCA elects as CH the sensor with the lowest ID among its neighbors that not received a CH announcement. Listing 1.6 shows the two states in which there are lines in the SLEDS program that differ from the LEACH code. Line 27 from Listing 1.3 has been removed and Lines 5 and 14 differ on the arguments to functions `selectCH` and `knownCHs.insert`.

```
1  Program Coordinator() {
2    list<int> knownNeighbors;
3
4    STATE Select_CH() {
5    if (compCH->selectCH(myID, knownNeighbors)) {
6      broadcast(CH_ANNOUNCE, GetNextMsgId(), myID);
7      compSensor->role = CH; }
8    else {
9      compSensor->role = CM; }
10   nextState Join_Cluster(); }
```

```

 STATE Join_Cluster() {
 During (tCluster) on recvBroadcast(CH_ANNOUNCE, msgID, ID) {
   knownCHs.insert(ID); }
 nextState Cluster_Formation(); }
```

Listing 1.6. The LCA Coordination

The two case studies presented in this section show that the model instances share the same state machine specification, promoting reusability. The programmer develops a few lines of code with the specificities of each model. Moreover, the state machine primitives adopted by SLEDS does not impose any fixed flow of activities (such as CBCWSN), but allow the developer to define the coordination of any data-centric model.

5 Conclusion

In this paper we proposed a Domain-Specific Language, called SLEDS, for prototyping Wireless Sensor Network applications that adopt a data-centric storage approach. The current specification of the language generates code to run on the NS2 simulation environment, using a library of components provided by RCBM [7]. We validate our approach by defining a syntax-directed translation into NS2 code. As case studies we developed SLEDS programs for LEACH and LCA data-centric models. Our experiments showed that both models share many similarities on the flow of activities. The achieved results show that SLEDS allowed code reuse and agile development for the LCA specification. Our proposal answers some of the challenges identified in [12]. In the future, we intend to implement the parser to translate SLEDS program to NS3 code, a more intuitive NS2 evolution as well as to other simulators and real networks.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2Nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
2. Amaxilatis, D., Chatzigiannakis, I., Koninis, C., Pyrgelis, A.: Component based clustering in wireless sensor networks. arXiv preprint arXiv:1105.3864 (2011)
3. Baker, D.J., Ephremides, A.: A distributed algorithm for organizing mobile radio telecommunication networks. In: Proceedings of the 2nd International Conference on Distributed Computing Systems, Paris, France, 1981. pp. 476–483 (1981)
4. Baumgartner, T., Chatzigiannakis, I., Fekete, S., Koninis, C., Kroller, A., Pyrgelis, A.: Wiselib: A generic algorithm library for heterogeneous sensor networks. EWSN: LNCS **5970**, 162–177 (2010)
5. Carrero, M., Zamproni, K., Musicante, M.A., Santos, A., Hara, C.: Uma máquina de estados para especificação de códigos de simulação para redes de sensores sem fio urbanas. In: Simpósio Brasileiro de Computação Ubíqua e Pervasiva (2018)
6. Carrero, M.A., da Silva, R.I., dos Santos, A.L., Hara, C.S.: An autonomic in-network query processing for urban sensor networks. In: 20th IEEE Symp. on Computers and Communications (ISCC). pp. 968–973 (2015)

7. Carrero, M.A., Musicante, M.A., dos Santos, A.L., Hara, C.S.: A reusable component-based model for WSN storage simulation. In: Proceedings of the 13th ACM Symposium on QoS and Security for Wireless and Mobile Networks. pp. 31–38 (2017)
8. Chandra, T.B., Dwivedi, A.K.: Programming languages for wireless sensor networks: A comparative study. In: 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom). pp. 1702–1708 (March 2015)
9. D'Angelo, G., Diodati, D., Navarra, A., Pinotti, C.M.: The minimum k-storage problem: Complexity, approximation, and experimental analysis. IEEE Trans. Mob. Comput. **15**(7), 1797–1811 (2016)
10. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. **35**(6), 26–36 (2000)
11. Diallo, O., Rodrigues, J.J.P.C., Sene, M., Mauri, J.L.: Distributed database management techniques for wireless sensor networks. IEEE Trans. Parallel Distrib. Syst. **26**(2), 604–620 (2015)
12. Estrin, D., Govindan, R., Heidemann, J.S., Kumar, S.: Next century challenges: Scalable coordination in sensor networks. In: Kodesh, H., Bahl, V., Imielinski, T., Steenstrup, M. (eds.) MobiCom. pp. 263–270. ACM (1999)
13. Fangfang, L., Zhibo, F., Chuanwen, L., Jia, X., Ge, Y., Shenyang, C.: A data storage method based on multilevel mapping index in wireless sensor networks. In: International Conference on Wireless Communications, Networking and Mobile Computing. p. 2747 –2750 (2007)
14. Fischer, J., Majumdar, R., Millstein, T.: Tasks: language support for event-driven programming. In: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. pp. 134–143. ACM (2007)
15. Fowler, M.: Domain Specific Languages. Addison-Wesley Professional, 1st edn. (2010)
16. Heinzelman, W.R., Chandrakasan, A., Balakrishnan, H.: Energy-efficient communication protocol for wireless microsensor networks. In: 33rd Annual Hawaii International Conference on System Sciences (HICSS-33), 4-7 January, 2000, Maui, Hawaii, USA (2000)
17. Lee, E.K., Viswanathan, H., Pompili, D.: Distributed data-centric adaptive sampling for cyber-physical systems. TAAS **9**(4), 21:1–21:27 (2015)
18. Louden, K.C.: Compiler Construction: Principles and Practice. PWS Publishing Co., Boston, MA, USA (1997)
19. Murr, F., Mauerer, W.: Mcfsm: Globally taming complex systems. In: Proceedings of the 3rd International Workshop on Software Engineering for Smart Cyber-Physical Systems. pp. 26–29. SEsCPS '17, IEEE Press, Piscataway, NJ, USA (2017)
20. Newton, R., Morrisett, G., Welsh, M.: The regiment macroprogramming system. In: 2007 6th International Symposium on Information Processing in Sensor Networks. pp. 489–498 (April 2007)
21. Salman, A.J., Al-Yasiri, A.: Developing domain-specific language for wireless sensor network application development. In: 11th International Conference for Internet Technology and Secured Transactions, ICITST 2016. pp. 301–308 (2016)
22. Shen, H., Zhao, L., Li, Z.: A distributed spatial-temporal similarity data storage scheme in wireless sensor networks. IEEE Transactions on Mobile Computing **10**(7), 982–996 (2011)
23. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: A neighborhood abstraction for sensor networks. In: Proceedings of the 2Nd International Conference on Mobile Systems, Applications, and Services. pp. 99–110. MobiSys '04, ACM (2004)