Development of Wireless Sensor Networks Applications with State-based Orchestration

Alexandre Rabello Ordakowski, Marco Aurélio Carrero and Carmem Satie Hara
Universidade Federal do Paraná, Curitiba-PR, Brazil

Keywords: Wireless Sensor Network, Application Development, Orchestration, State Machine.

Abstract: The growing demand for sensor devices, key elements of cyber-physical systems and the Internet of Things, requires fast development of new applications. However, the specification and implementation of such systems is a complicated task, especially because of the lack of support for code reuse and for defining the program execution flow. Service orchestration is a technique that has been widely adopted for developing applications for the cloud. In this paper we propose a similar technique for developing applications for Wireless Sensor Networks (WSN). To this end, we propose a development model based on reusable software components for WSN applications. For the components orchestration, which defines the application execution flow, we propose a domain-specific language, called SLEDS-SD (State Machine-based Language for Sensor Devices). In its current implementation, SLEDS-SD generates nesC code, which can be installed in TinyOS-based devices. The evaluation involved the development of three cluster-based WSN models. The efficiency of the proposal was evaluated by determining the amount of code reuse, while its efficacy was evaluated by the generated code correctness. For that, we compare the generated programs behavior with those reported in previous studies.

1 INTRODUCTION

Cyber-physical systems and the Internet of Things (IoT) rely on sensor devices to capture information from the environment and application users. They are key components that compose the data acquisition layer of such systems. A common technique, advocated by edge (Satyanarayanan, 2017) and fog computing (Dastjerdi and Buyya, 2016), is to process and store data either on or close to the source. In line with this trend, in addition to the increase of storage and processing capacities of current sensor devices, we can envision a system architecture in which sensors play an important role not only for data acquisition, but also as active components for storing and processing data.

One approach to define the interconnection among the devices is by orchestration, which determines the application workflow (Adel Serhani et al., 2020). The complete IoT application not only involves the orchestration of the services provided by the devices, but also the implementation of the services and, in particular, of the sensoring component of IoT devices. Sensors are fundamental components of IoT. Thus, there is a need to investigate development models for wireless sensor networks (WSNs) as well. However, the development of such systems is a complex task. It is observed that few studies in the literature propose an approach that supports the implementation of WSN applications considering both orchestration and code reuse. To address this problem, we propose an orchestration approach to develop the sensor subsystem of IoT applications. Such a sub-system may involve both data storage and processing. In order to reduce the time spent to develop new applications, our model also promotes code reuse, based on the concept of software components. We consider three types of components: application components, which are related to the application's tasks; library components, which provide generic functions; and the coordination component, responsible for orchestrating the program execution flow.

The orchestration is designed and implemented as a state machine, which is a model adopted by several previous works (Taherkordi et al., 2015; Braga, 2012; Lekidis et al., 2018; Hussein et al., 2017). However, in contrast to these works and traditional sensor and network simulator languages (such as nesC and NS-2 and 3), which target only event-based transitions, our state machine supports both event-based and logic-based transitions. Event-based transitions may be adequate to model the high-level behavior of the application, but they are not sufficient to model the internal

Ordakowski, A., Carrero, M. and Hara, C.
Development of Wireless Sensor Networks Applications with State-based Orchestration.
DOI: 10.5220/0010499703120319
In *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*, pages 312-319
ISBN: 978-989-758-510-4

behavior of a sensor device, which is the focus of our work. Logic-based transitions support the design and development of complex data processing tasks, rather than only data acquisition.

In order to support this orchestration model, and allow the implementation of programs directly from the state machine specification, we have designed a language, called SLEDS-SD (State Machine-based Language for Sensor Devices), which supports the concepts of states and transitions. We envision that in the future, the same SLEDS-SD program can be used to generate code for several languages, in order to be installed on devices with heterogeneous platforms. In this paper, we report on the translation of SLEDS-SD programs to nesC (Gay et al., 2003), the language used to develop programs for the TinyOS platform. We have chosen this language because it was developed for resource constrained devices, and avoids dynamic typing and memory allocation. Thus, if we are able to generate nesC code from SLEDS-SD, most likely we will be able to generate code for other modern languages with more expressive power.

SLEDS-SD extends the work proposed in (Carrero et al., 2021), which targets the development of *simulation* code in NS-2. Here, by targeting a resource constrained language, to be installed in real sensor devices, both the development model and the language were modified.

In order to evaluate our proposal, we considered the development of three cluster-based WSN models. Its efficiency was evaluated by determining the amount of code reuse, which measures the reduction of development effort. The efficacy was evaluated by determining the generated code correctness. For that, we compare the generated programs behavior with those reported in previous studies.

Paper Outline: Section 2 discusses related works. In Section 3 our development model is presented. The language SLEDS-SD is described in Section 4, while Section 5 reports the results of the experimental evaluation. Final considerations are presented in Section 6.

2 RELATED WORK

This section discusses existing works in the literature with similar goals as our development model. SenNet (Salman and Al-Yasiri, 2016), IoTSuite (Patel and Cassou, 2015) and WiseLib (Baumgartner et al., 2010) adopt different design models. In **SenNet** models are defined based on the sensor's functions in the network topology, such as nodes and clustering. However, as opposed to SLEDS-SD, it does not adopt a flexible execution model. **IoTSuite** adopts a high-level language that allows syntax customization according to the IoT application domain. This approach provides generality, but requires the work of a specialist to configure a new language for each domain. **Wiselib** is a framework composed of a set of generic libraries to generate code for different platforms, such as TinyOS and ScatterWeb. Although it proposes an efficient method of code reuse, it does not adopt a flexible coordination execution flow.

Tokenit (Taherkordi et al., 2015), X-Machine (Braga, 2012), Hussein's model for IoT (Hussein et al., 2017) and RCBM (Carrero et al., 2021) are models based on state machines. In **Tokenit** the programmer develops the activities in C, while the execution flow is specified in XML. Although Tokenit separates the execution flow from the activities flow, it requires some effort to describe the XML model and then to manually interleave the generated code. Our focus relies on providing services as a component, and a language close to the specification model to generate the execution flow. In **X-Machine**, *states* represent hardware components, and are directly associated with their functionalities. Thus, it targets applications on the hardware level, while we consider states in the software level. Another state machine model to represent different system configurations with event-based transitions, such as hardware failures, is proposed by (Hussein et al., 2017). However, their focus is on system adaptability, while ours is on code reuse of software components.

Our proposal shares with **RCBM** the same principles of code reuse and a language based on state machines. In fact, SLEDS-SD is an adaptation of their proposed SLEDS language, by taking into consideration a resource constrained language for sensor nodes, such as nesC. This is because SLEDS was designed to generate NS-2 *simulation* code, which is based on C++ and thus supports dynamic typing. This feature gives better support for component-based software development, but usually does not exist in languages for programming sensors. As a result, SLEDS-SD modifies all constructs in the language that relies on dynamic typing. Moreover, SLEDS-SD promotes a top-down development of application components because the translation process from SLEDS-SD to nesC also generates a set of components interfaces that can later be implemented by the programmer. We intend in the future implement a translation from SLEDS-SD to several other IoT platforms. By doing so, the same SLEDS-SD code can be deployed on heterogeneous devices that compose the system.

3 WSN APPLICATION DEVELOPMENT MODEL

The development model proposed in this paper is based on orchestration of software components using a state machine, and development of components to implement specific tasks. The three steps of the model are depicted in Figure 1: (1) specification using a state machine, (2) development of an orchestration program in SLEDS-SD and development of application components, and (3) code generation in a target sensor language and system deployment.

3.1 Specification Step

In the specification of systems for WSNs, a widely used model is their representation as state machines (Ouadjaout et al., 2014), given the reactive nature of sensor devices. However, previous state-based works consider only event-based transitions, such as the receipt of a message. This type of transition may be sufficient to model the general state of a device, such as light on/off. However, our previous experience in developing WSN applications showed that it is not sufficient to model the internal behavior of the sensor to implement more complex data processing tasks. Such tasks are now possible and desirable, given the processing and storage capacity of modern sensors. Based on this observation, our state machine also supports logic-based transitions. More specifically, we can specify the execution flow of a data processing task as a state machine, and thus a sensor may be in a different state of the task at any given time. It allows, for example, to specify different reactions of the sensor on the receipt of a message, based on its current state. States in our model define logical steps for accomplishing a task, which may involve collaboration among sensors, triggered by message interchanges, and possibly controlled by a timer.

Thus, there are two types of event-based transitions: triggered by a request, such as the receipt of a message, and triggered by a timer timeout. We consider that all sensors in a WSN are executing the same application, although each of them may be in a different state at a given time. They communicate via radio, through multi-hop message passing. Thus, whenever a sensor sends a message, all neighbors within its radio transmission range receives it. However, it is important to note that considering each sensor device individually, there is no relation between the state in which a message is sent and the state in which it receives it. That is, if a sensor sends a message from state *s*, it is not always the case that its neighbors are in the same state *s*. In fact, each of them may be in

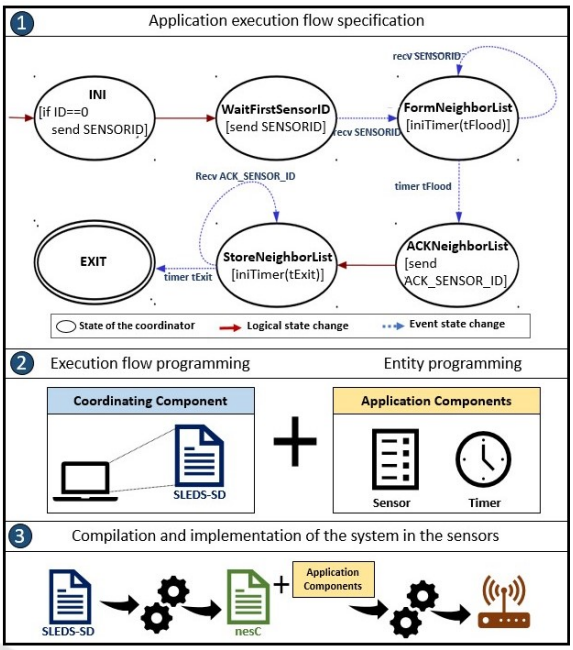

Figure 1: Development model.

a different state, and react to the message in a different way. As a result, in our model, states have a name and may be associated with two types of actions: send a message or initiate a timer. Receiving a message (or a timer timeout) triggers an event-based transition that do not necessarily start from the state that sent the message (or that initiated the timer). Event-based transitions are labeled with their triggering event (message type or timer), while logical transitions may be labeled with a logical expression.

An example is shown in Figure 1(1). The state machine computes, for each sensor, its set of neighbors. It starts with state *INI*, in which the sensor with ID 0 sends a `SENSOR_ID` message in broadcast. All sensors make a logical transition (depicted in red) to state *WaitFirstSensorID*. In this state, the receipt of the first `SENSOR_ID` message triggers the sensor to broadcast its own `SENSOR_ID` message and make an event-based transition (depicted in blue) to state *Form_Neighbor_List*. In this state, a new timer `tFlood` is initiated, and during this time it keeps receiving `SENSOR_ID` messages to create its list of neighbors. Upon `tFlood` timeout, an event transition is made to state *ACK_Neighbor_List*. In this state, sensors send a confirmation message to their neighbors and make a logical transition to state *StoreNeighborList*. In this state, sensors receive `ACK_SENSOR_ID` messages from their neighbors and store their IDs during time `tExit`. Upon its timeout, the application make a transition to the *EXIT* final state.

3.2 Programming Step

Based on the state machine specification, a corresponding program is developed in our language, called SLEDS-SD. A SLEDS-SD program consists of a set of states, which in turn, consists of a sequence of actions. An example of a state implementation can be found in Listing 1. It corresponds to state *FormNeighborList* in Figure 1(1). Observe that *during (timer) on recvBroadcast* is the command in SLEDS-SD that indicates the initialization of a *timer* and the receipt of messages during this period of time. In the example, during *tFlood* time, the application receives messages of type *SENSOR_ID*. Upon each receipt, the sensor *id* received in the message is stored in a list *sensorList*. Finally, when the timer expires, a transition is made to *AckNeighborList* state. It is worth noticing the similarity between the state machine specification and the constructs of the language, which will be detailed in Section 4.

```
STATE Form_Neighbor_List() {
 during (tFlood)
  on recvBroadcast(SENSORID, msdID, pktSensorId){
    sensorList.insert(pktSensorId.id);}
    nextState ACK_Neighbor_List()}
```

Listing 1: Form_Neighbor_List state in SLEDS-SD.

While programming the states, some functions provided by existing components or yet-to-be implemented components may be called. Typically, there are two types of components: library components and application components. Library components provide generic functions. Some languages, such as nesC, already provide some, like the timer component. It can be used to generate events at regular intervals of time. Additional library components can be implemented, such as the data aggregation component, to provide functions, like sum, average, maximum and minimum, to be computed from the values in a list. Application components are directly associated to entities in the application domain. As an example, in a cluster-based WSN model, application components may include sensor, gateway and cluster. Each component provides a set of functions. For example, the cluster component may have functions for choosing the cluster-head (CH), where the CH is the sensor responsible for storing cluster members information and answering query requests.

In SLEDS-SD, programs that use components must contain explicit *use* statements at the beginning of the program. Library components are ready to be reused by any application. Application components, on the other hand, may or may not have been previously implemented. To support the idea of reusing existing components, while allowing the definition of new ones, and still adhering to a static typed target sensor programming language, our model uses the following strategy. If the component or its interface exists, types are checked, in the same way as traditional compilers. However, if they do not exist, a header file with their interfaces is generated to guide the developer on how to complete the application. This strategy, while promoting code reusability, helps the developer determine which functions should be implemented for each application component.

3.3 Code Generation and Deployment

In the last step, SLEDS-SD code is compiled and translated to a target sensor programming language. In our current implementation, we generate code in nesC. Thus, the SLEDS-SD compiler generates nesC code that corresponds to the specified state machine, along with a header file for each application component. The set of files are given as input to the nesC compiler, which generates the executable code to be deployed on the sensor devices.

Our model simplifies the development of WSN applications with a clear separation between the specification and implementation, while helping the implementation with a high-level programming language that closely resembles the specification model. Moreover, it promotes code reuse, with the creation of connectable components.

4 THE SLEDS-SD LANGUAGE

This section presents the SLEDS-SD language and its translation to nesC. Most of its constructs derive from SLEDS (Carrero et al., 2021), a language proposed to generate *simulation* code for NS-2. Thus, in this section we focus on the main constructs of the language and the changes made in order to allow its translation to nesC.

```
use compSensor as ComponentSensor;
use compLibMSG as ComponentLibMessage;

program MaxMinCoordinator () {
 const tCluster=25;
 int round, myID, winnerID, myCH;
 list <int> sensorList[5];

 state Ini(){
  winnerID=compSensor->getSensorId();
  nextState FloodMax(winnerID); }
```

Listing 2: Structure of a SLEDS-SD program.

A SLEDS-SD program consists of three parts: (i) component import declarations; (ii) data declarations; and (iii) state definitions, as depicted in Listing 2. Among the data declarations, it is required that the structure of messages transmitted among sensors, as well as their identifiers be defined using the syntax shown in Figure 2. Listing 3 provides examples of these declarations, along with the declaration of a variable of type `msgSensorId`. The **message type** declaration is similar to an enumeration type in C and defines a set of message type identifiers. They help the implementation of events that receive messages, allowing different types of messages be handled properly. **message** declarations define new types in the language, with the structures of the messages. The language primitives for exchanging messages (*send*, *broadcast*, *recv* and *recvBroadcast*) have a parameter of this type.

```
messageType::= MESSAGE TYPE '{' identifier ( ',' identifier )* '}' ';'
message::=     MESSAGE identifier '{' typeList '}' ';'
```

Figure 2: Message type and message structure syntax.

```
message type{SENSORID, MAXWINNER, MINWINNER};

message msgSensorId { int id; };
message msgMaxWinner { int maxWinnerId; };
message msgMinWinner { int minWinnerId; };

msgSensorId pktSensorId;
```

Listing 3: Message type and message declarations.

All programs must have an `Ini` state, where the execution starts, and one or more final states. The language primitives to implement states include the traditional control flow statements, such as *if*, *for* and *while*, as well as ones concerning events and state transitions[1]:

- **nextState:** makes a transition to another state;
- **broadcast:** sends a message in broadcast;
- **send:** sends a message to a specific sensor;
- **on recvBroadcast:** receives a broadcast message;
- **on recv:** receives a message addressed to a specific sensor;
- **during:** triggers a timer.

The language also supports composite actions. They have been identified as commonly used for developing applications for WSNs. They define that a sensor waits for messages of a given type during a period of time:

- **during** (*t*) **on recvBroadcast** (*message*) {*ActionList*} **nextState** *state*: composite action for receiving broadcast messages during time *t*, followed by a state transition.
- **during** (*t*) **on recv** (*message*) {*ActionList*} **nextState** *state*: composite action to receive messages addressed to the sensor itself during time *t*, followed by a state transition.

A state with a composite action is shown in Listing 1. In this example, we can now see that the parameters for `recvBroadcast` are: the message type, a unique message identification, and a message variable, as declared in Listing 3. Statements `recv`, `broadcast`, and `send` have the same parameters. Moreover, variable `sensorList` has been declared in Listing 2 as a list of type `int`, which has `insert` among its associated functions. Definitions of lists of all atomic types in the language as well as message types were among the extensions needed in the SLEDS language in order to enable the translation to nesC.

4.1 Translation to nesC

The translation from SLEDS-SD to nesC generates one nesC method for each state in the source code. The nesC method and the corresponding SLEDS-SD state code are quite similar when they do not involve events (timer and message receipt). However, the translation is not so simple when events are present. In this case, parts of the state code have to be generated in different parts of the nesC program. This is because the end of a timer and the receipt of messages in nesC are considered asynchronous events with respect to the execution flow of the application or the current state. Thus, nesC requires all messages to be handled by a single `receive` event. Moreover, a timer timeout triggers the execution of an associated `fired` event. Thus, the translation from SLEDS-SD to nesC requires spreading the code of a SLEDS-SD state into both the corresponding nesC method and the events. Moreover, in order to perform the actions associated to these events defined in each SLEDS-SD state, we introduce the variable *currentState* in the nesC code to store the sensor's processing state.

To exemplify the process, consider the state machine that specifies part of the clustering algorithm of the MAX-MIN model (Amis et al., 2000) depicted in Figure 3. This model adopts a very distinctive flow of execution, with multiple flooding rounds to collect information from the neighbors, and then, based on this information, creates clusters, selects cluster-heads and gateways. Consider the state *StoreMaxWinner*, shown in Listing 4, which contains a `during-on recv` composite statement. Its trans-

[1]The complete SLEDS-SD language specification can be found at https://github.com/sleds-sd.

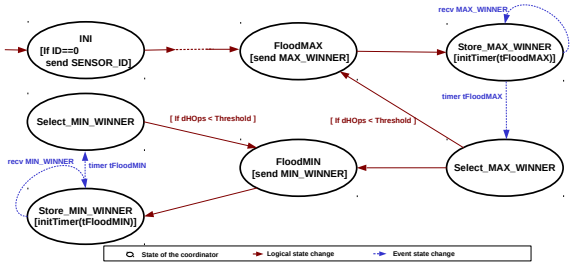

Figure 3: FloodMax and FloodMin State Machine.

lation to nesC is presented in Listing 5. Observe that the code is generated in 3 different parts. The first block of code is generated in the corresponding *state_StoreMaxWinner* method. In this method, the current state is stored in the *currentState* variable, by assigning the value *STOREMAXWINNER*. In addition, it starts a timer, which corresponds to the *during* statement. The second block of code is generated in the **receive** event. This event manages all messages received by the sensor. Thus, this event is mainly composed of a sequence of `if-else if` statements that checks the value of the `currentState` variable, and includes in this portion of the program the list of actions within the SLEDS-SD `on recv` instruction. Similarly, the third block of code is generated in the timer **fired** event, with a similar structure. That is, it checks whether variable *currentState* contains the value *STOREMAXWINNER* and the associated action is to make a state transition, by calling the method *state_SelectMaxWinner*.

```
STATE StoreMaxWinner ( ) {
 during (tFloodMAX)
 on recv(MAXWINNER, msgID, pktMaxWinner) {
   ListMaxWinner.insert
   (pktMaxWinner->winnerID); }
 nextState SelectMaxWinner (); }
```

Listing 4: StoreMaxWinner state in SLEDS-SD.

Spreading the code in different methods and events make the implementation in nesC a hard task. A program in SLEDS-SD, on the other hand, supports the development of the program as a state machine, allowing the entire logic of each step to be coded within a single state. As a result, SLEDS-SD programs are more compact and easier to develop and maintain.

```
void state_StoreMaxWinner() {
 currentState = STOREMAXWINNER;
 call Timer.startOneShot(tFloodMAX); }

event message_t* Receive.receive(message_t*
msg, void* payload, uint8_t len) {
 [...] //suppressed code
 } else if(currentState == STOREMAXWINNER){
  msgMaxWinner*pktMaxWinner=
  (msgMaxWinner)payload;
  call compList.insert(ListMaxWinner,
    pktMaxWinner->winnerID);
 } [...] //suppressed code
 return msg; }

event void Timer.fired() {
 [...] //suppressed code
 }else if(currentState == STOREMAXWINNER) {
  state_SelectMaxWinner( );
 } [...] //suppressed code }
```

Listing 5: NesC code of StoreMinWinner state.

5 EXPERIMENTAL STUDY

We have implemented a translator from SLEDS-SD to nesC using Flex and Bison for developing the lexical and syntactical analyzers, respectively. The translator, as well as the applications code presented in this section, are available in a repository on GitHub[2].

In order to evaluate our approach, we applied the development model to implement three cluster-based WSN models: LCA (Baker and Ephremides, 1981), LEACH (Heinzelman et al., 2000) and MAX-MIN (Amis et al., 2000). In these models, some sensors in the network are elected leaders, called *cluster-heads* (CH), and they are responsible for storing the readings of a group of sensors (*cluster-members*) that compose a *cluster*. Table 1 shows the main characteristics of each model. LCA and LEACH have been chosen because they follow a traditional clustering execution flow. As a result, our experimental analysis is relevant for determining the impact of our model when developing a new system with an execution flow similar to one that has already been implemented following our approach. On the other hand, MAX-MIN was selected because it involves a very distinctive execution flow, as illustrated in Figure 3.

We have conducted two experiments: *(i)* to determine the impact of code reuse and *(ii)* to validate the correctness of our implementation by applying it to the same evaluation scenario and parameters reported in (Amis et al., 2000).

Table 1: Main features of compared models.

Model	Cluster-Head Election	Cluster Formation
LCA	The lowest ID	CH with the highest radio signal intensity
LEACH	Probabilistic divided into rounds	CH with the highest radio signal intensity
MAX-MIN	The lowest ID in the list of highest IDs	The first CH on the Gateway route

[2]SLEDS-SD repository: https://github.com/sleds-sd

Table 2: Number of states, lines and reused code.

Model	#states SLEDS	#lines SLEDS	#lines nesC	Reused lines
Coordinator				
LCA	7	92	153	99.48 %
LEACH	7	98	164	95.97 %
MAX-MIN	14	140	233	24.28 %
Components				
LCA			113	78.76 %
LEACH			119	74.78 %
MAX-MIN			242	32.32 %

5.1 Reusability Analysis

Table 2 presents the the amount of code reuse promoted by our development model. Recall that in our model the flow of execution is implemented in SLEDS-SD, while the components are implemented in nesC. Here, we denote the SLEDS-SD program as the *Coordinator*. As the LCA and LEACH models adopt similar execution flows, their coordinators contain more than 95% identical lines of code in SLEDS-SD. The code differs only to take into consideration distinct criteria for the CH election. For MAX-MIN, however, the percentage of identical lines is around 24%, given its distinctive execution flow. Turning to the implementation of components, the similarity of nesC lines of code for LCA and LEACH is around 75%, while for MAX-MIN it is 32%. Here the similarity follows from the existence of some functions in the components that are identical in all models, such as computing the set of neighbors for each sensor device (as illustrated in Figure 1(1).

Table 2 also shows, for each model coordinator, the number of states, number of lines of code in SLEDS-SD, and number of lines of code in nesC generated by the translator. LCA and LEACH contain 7 states, and 92 and 98 lines of SLEDS-SD code, respectively. MAX-MIX has 14 states, with 140 lines of SLEDS-SD code. We can notice that the SLEDS-SD program is roughly 40% smaller than the one generated in nesC by our translator. This suggests that the development effort for SLEDS-SD is also smaller than for nesC.

These results show that our approach can potentially reduce the time spent to develop new WSN applications, both by reusing a coordinator with a similar execution flow when it exists, and by reducing the development effort with a high-level language that closely resembles the specification model.

5.2 Generated Code Correctness

The purpose of this experiment is to determine the correctness of the code generated by our translator from SLEDS-SD to nesC. To do so, we compare the behavior of the generated code with those reported by (Amis et al., 2000). The simulation parameters are also the same as the ones adopted by (Amis et al., 2000). It considers a 200x200 square meters monitored region, and network densities of 100, 200, 400 and 600 sensor devices. The radio range of every sensor on the field was set to 20 meters. The results presented in this section correspond to the average of 35 simulations executed on TOSSIM, the TinyOS simulator, with a confidence interval of 95%.

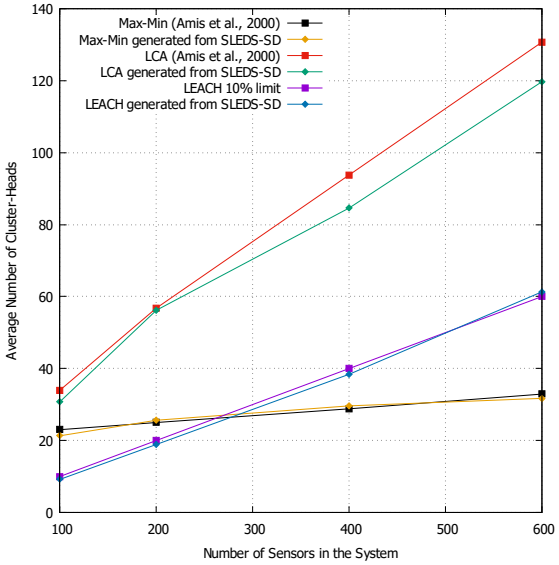

Figure 4: Impact of network density on number of CHs.

Figure 4 shows the number of CHs for each network density, measured both by the code generated from SLEDS-SD and reported by (Amis et al., 2000). The graph shows that the values are quite similar and that they present a consistent growth ratio with the increase of the network density. The small differences are due to the network topology, that may not be identical in both works. In our experiments, for each of the 35 executions and each network density, we randomly generated the position of the sensors on the field. For LCA and MAX-MIN, the maximum number of wireless hops between a node and its cluster-head was set to 2. For LEACH, the number of CHs is defined as a percentage of the number of devices, which was set as 10%. In the graph the exact 10% of of sensors is presented in purple, while the number generated by SLEDS-SD is in blue. It closely follows the expected value, with 38.4 sensors elected as CHs for a network density of 400 (9.6%) and 61.3 sensors elected as CHs for a network density of 600 (10.21%). These results show that our strategy for translating SLEDS-SD code to nesC is correct and does not impact the quality of the generated code.

6 CONCLUSION

In this paper we proposed a development model for WSNs based on state machines. The goal was to support the implementation of new applications based on two principles: (i) specification of the flow of execution as a state machine, and (ii) implementation of components that provide reusable services. We introduced a language, called SLEDS-SD, and developed a translator from SLEDS-SD to nesC in order to generate code to be installed on devices of real WSNs. We presented an experimental study that shows that SLEDS-SD programs are about 40% smaller than the generated nesC code, which suggests that our approach reduces the development effort. Moreover, the evaluation of code reuse showed that applications with a similar execution flow have more than 95% of identical lines of SLEDS-SD code. The rate of code reuse of components implemented in nesC ranged from 32% to 78% following our approach.

As future work, we intend to extend our development model to generate code for both network simulators, such as NS-3, as well as other WSN and IoT platforms, such as Contiki (Dunkels et al., 2004) and RIOT (Baccelli et al., 2013). With such extensions it will be possible to use the same high-level specification to first evaluate a clustering model using simulators and then deploy the application in different real platforms with much less effort. We also intend to extend the experimental study by analyzing the generated code quality, both in terms of legibility as well as of resources utilization by the sensor devices. Another lines of investigation include support for searching reusable components, and support for two-level orchestration: on device-application level and on application internal behavior level.

ACKNOWLEDGEMENTS

This work was partially funded by CNPq and CAPES-PrInt-UFPR.

REFERENCES

Adel Serhani, M., El-Kassabi, H. T., Shuaib, K., Navaz, A. N., Benatallah, B., and Beheshti, A. (2020). Self-adapting cloud services orchestration for fulfilling intensive sensory data-driven iot workflows. *Future Generation Computer Systems*, 108:583 – 597.

Amis, A. D., Prakash, R., Vuong, T. H., and Huynh, D. T. (2000). Max-Min D-Cluster formation in wireless ad hoc networks. In *Proc. of the IEEE Conf. on Computer Communications*, volume 1, pages 32–41.

Baccelli, E., Hahm, O., Günes, M., Wählisch, M., and Schmidt, T. C. (2013). Riot os: Towards an os for the internet of things. In *IEEE Conf. on Computer Communications Workshops*, pages 79–80.

Baker, D. J. and Ephremides, A. (1981). A distributed algorithm for organizing mobile radio telecommunication networks. In *ICDCS*, pages 476–483.

Baumgartner, T., Chatzigiannakis, I., Fekete, S., Koninis, C., Kroller, A., and Pyrgelis, A. (2010). Wiselib: A generic algorithm library for heterogeneous sensor networks. In *European Conf. on Wireless Sensor Networks*, pages 162–177.

Braga, M. d. L. (2012). Geração automática de código para redes de sensores sem fio usando communicating x-machine. Master's thesis, Programa de Pós-graduação em Engenharia Elétrica - UFAM, Manaus.

Carrero, M. A., Musicante, M. A., dos Santos, A. L., and Hara, C. S. (2021). A DSL for WSN software components coordination. *Information Systems*, 98:101461.

Dastjerdi, A. V. and Buyya, R. (2016). Fog computing: Helping the internet of things realize its potential. *Computer*, 49(8):112–116.

Dunkels, A., Gronvall, B., and Voigt, T. (2004). Contiki-a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE Int. Conf. on Local Computer Networks*, pages 455–462.

Gay, D., Levis, P., Von Behren, R., Welsh, M., Brewer, E., and Culler, D. (2003). The nesc language: A holistic approach to networked embedded systems. *Acm Sigplan Notices*, 38(5):1–11.

Heinzelman, W. R., Chandrakasan, A., and Balakrishnan, H. (2000). Energy-efficient communication protocol for wireless microsensor networks. In *Proc. of the 33rd Annual Hawaii Int. Conf. on System Sciences*. 10 pages vol.2. IEEE.

Hussein, M., Li, S., and Radermacher, A. (2017). Model-driven development of adaptive iot systems. In *MODELS (Satellite Events)*, pages 17–23.

Lekidis, A., Stachtiari, E., Katsaros, P., Bozga, M., and Georgiadis, C. K. (2018). Model-based design of iot systems with the bip component framework. *Software: Practice and Experience*, 48(6):1167–1194.

Ouadjaout, A., Lasla, N., Bagaa, M., and Badache, N. (2014). Static analysis of device drivers in tinyos. In *Proc. of the 13th Int. Symposium on Information Processing in Sensor Networks*, pages 297–298.

Patel, P. and Cassou, D. (2015). Enabling high-level application development for the internet of things. *Journal of Systems and Software*, 103:62–84.

Salman, A. J. and Al-Yasiri, A. (2016). Sennet: a programming toolkit to develop wireless sensor network applications. In *Proc. of the 8th IFIP Int. Conf. on New Technologies, Mobility and Security*, pages 1–7.

Satyanarayanan, M. (2017). The emergence of edge computing. *Computer*, 50(1):30–39.

Taherkordi, A., Johansen, C., Eliassen, F., and Römer, K. (2015). Tokenit: Designing state-driven embedded systems through tokenized transitions. In *Int. Conf. on Distributed Computing in Sensor Systems*, pages 52–61.