A Distributed Compositional Language for Wireless Sensor Networks

D. Janakiram and R. Venkateswarlu
Department of Computer Science and Engineering
Distributed and Object Oriented Systems Lab, IIT Madras.
Chennai, India 600036
{djram, rv}@cs.iitm.ernet.in

***Abstract*— The paper presents a "Distributed Compositional Language" for wireless sensor networks that represents a new programming paradigm for application developers. This language simplifies the process of implementing applications in sensor networks. The emphasis, in this language is on styles. The styles specify the interaction of components. The paper shows, how styles such as Event style, Pipe style, Group Communication style can be used in sensor networks. This paper also presents implementation of monitoring and target tracking applications using this language.**

Keywords:Programming languages, Mate, Language design, Middleware, Paradigm, Components, Styles, Sensor networks, NesC, TinyScript

I. Introduction

Wireless sensor networks randomly deploy tens to thousands of sensor nodes. Each sensor node has a separate sensing, processing, storage and communication unit. Wireless sensor networks have many possible applications in the scientific, medical, commercial, and military domains. Examples of these applications include environmental monitoring, smart homes and offices, surveillance, intelligent transportation systems, and many others.

Current wireless sensor network applications are being developed by using nesC, Tiny Script etc. Developing applications for wireless sensor networks is a tedious process. This is because, memory and energy are constrained and due to lack of suitable programming abstraction. Current sensor nodes(berkeley motes) have 8 to 128 KB of instruction memory and 512B to 4KB of RAM [1]. So, providing a language support in application development for sensor networks is vital.

Most approaches to building programming languages for sensor networks are for specific scenarios either in the number/type of sensor nodes used or the flexibility of the application being developed. There is a need for a suitable programming abstraction that can in corporate a wide range of scenarios and provides for extensibility and flexibility.

Currently there exists two languages, Mottle and Tiny Script, which work on top of MATE virtual machine. But these languages can't implement complex applications like target tracking because the languages do not have enough grammar to meet the needs of applications [2]. These languages are still in the developing phase. NesC is also used for programming sensor networks, but developing components is a tedious process [1]. One more difficulty in those languages is, if we want to capture interaction between components, which resides in different sensor nodes then the application developer has to take care of low level communication mechanisms. So, realizing applications is a tedious process. But in Distributed Composition Language, we can directly capture interaction between components, which resides in different sensor nodes. That is, the application developer is not aware of low level communication mechanism. In order to provide higher level of communication abstraction, there is need for developing Distributed Compositional Language for wireless sensor networks. The goal of our language is ”ease of implementing applications” in sensor networks. The proposed flexibility (ease of implementing applications) of a language has been illustrated with two applications in section IV.

The component interactions is the key issue in Distributed Composition Language. These components are basically small software entities. This means, the design of the component is simple. Software components are black-box abstractions.

Distributed Compositional Language must address the unique challenges like driven by interaction with environment, limited resources,etc.
Driven by interaction with environment: Sensor nodes are fundamentally event-driven, reacting to changes in the environment instead of driven by interactive or batch processing.
Limited resources: Sensor nodes have very limited physical resources, due to the goals of small size, low cost, and low power consumption. Some languages fail to handle this limitation because they may not produce optimized code.

We are building a Distributed Compositional Language that provides styles such as event style, pipe style, etc. Which are appropriate to handle applications exploiting the event-driven nature of sensor nodes. To some extent, the distributed compositional language produces optimized code, in order to reduce power of the sensor node.

In this paper we present implementation of Monitoring and Target tracking applications using our language. We also present the implementation of various styles.

To develop compositional language in a distributed environment (e.g, sensor networks), we need to use some kind of communication mechanism to interact with other components residing in different sensor nodes in the network. Currently, we have used a distributed shared object space as a communication mechanism (middleware).

The remainder of the paper is organized as follows. Language design issues, frame work and runtime environment are explained in section II. Section III presents specification of styles and also explains the usage of styles in wireless sensor networks. In section IV, we discuss monitoring and target tracking applications. Section V presents the survey of related work. Section VI concludes this paper with an outlook on future work. In appendix, we present algorithms for realizing styles (pipe, event).

II. Language Design

This section explains proposed frame work, characteristics of distributed compositional language and existing language design issues with an outlook on runtime environment.

Language design issues : This Distributed Compositional Language is already used for developing applications in distributed and mobile systems[3]. For these applications, we used a distributed shared object space as a middleware [4]. This section focuses on language issues, section III and section IV explains the use of language in sensor networks.

Software systems can be viewed in two distinct ways. A running system can be seen as a collection of interacting entities. However, we can also view the system as a composition of various software components [5].

To specify systems consisting of interacting components a Distributed Composition Language is required. The key challenge will be in defining a set of operators in the language, which represent different coordination styles.
Scripts are high level specifications of how the components have to be composed. Thus applications can be specified as

"Applications = Components + Scripts" [6].

Every component provides a set of services and may also need a set of required services. Components communicate with each other through well defined ports(methods). An architectural style defines a grammar consisting of component and connector types and a set of rules how components and connectors can be combined. According to a given application in sensor networks, we can define the plugs each component may have and the connectors that can be used to compose them. The plugs are the interfaces to a component, also called as service ports(methods).

Composition is a mechanism by which the components are connected and the style is the way in which the components interact. In general, a component is not designed in an isolation, but as part of a frame work for collaborating components. With this kind of collaborating framework, the components can be easily deployed and used.

Grammar poses a special role in the language development. A scanner and the parser are developed for the language using the generator tools JLex and CUP. For each style a syntax is defined and the syntax rule is expressed as sentential form in the grammar.

The characteristics of a Distributed Composition Language are

- Components as Abstractions: Components are software abstractions that are composed in various ways to yield applications. Component may be a module or an object.
- Plug Compatibility: The components should be designed in such a way that, many styles can use those components. It means plug compatibility of components increases.
- Scalability: the use of the language should scale from small to large systems.

A. Framework

The frame work shown in figure 1. The above layers that are part of our frame work are briefly explained below:

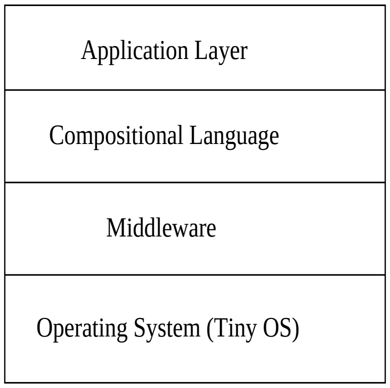

Fig. 1. Framework

- Application Layer: Programs should be written using components and styles provided by the language.

- Compositional Language: The scripts are passed to the lexical analyzer of the language which checks for the syntax errors and then it is passed to the parser, which in turn produces the code and that code is executed by Tiny OS.
- Operating System: TinyOS[7] operating system is used.

B. Runtime Environment

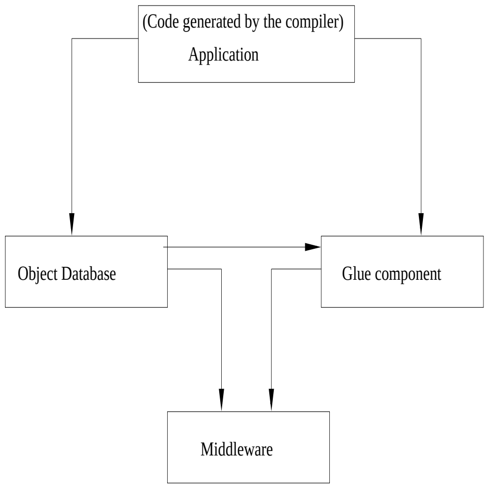

Fig. 2. Runtime Environment

The Runtime Environment mainly consists of

- Object Database
- Glue-Component
- Middleware

Object Database: This is used as a database. This stores all the symbols defined in the script (program). It also stores the mapping of the symbols associated with the components. So, whenever a symbol is referred a request to Object Database is sent. It searches for the component in the hash table on the symbol as a key and the search result is returned. This also instantiates the referred components and the object references are stored for future use. This is required as there is no exclusive construct in the language to instantiate the objects.

Glue-Component: The Glue-Component provides the glue code required in the language to implement and support different styles. The implementation of the semantics of the styles are done in the Glue-Component. GlueComponent also refers to the Object Database to resolve the symbols. This is used both at compile time and runtime. At compile time the compiler uses interface of the Glue-Component to verify the semantics of the styles. At run time the Glue-Component provides different interfaces for each style where the component interactions are implemented.

Middleware : Glue-component and Object Database uses the interfaces provided by middleware. Now we are designing a middleware for sensor networks. A distributed shared object space can be used as a middleware.

III. How to specify styles?

This section explains how to declare components and how to specify the styles.

def Component com1 = login

In above declaration, def and Component are keywords of language, whereas login is a component. When parser encounters a def keyword, it understands that the statement is a declaration of component. The com1 is a variable associated with login component.

run: com1.transfer() $\rightarrow$ com2.verify()

In above declaration, assume com1, com2 are two components. $\rightarrow$ arrow is called operator or connector and run is a keyword. The above style specifies the interaction of components as component2 is receiving data from component1.

Event Style: It is truly asynchronous in notifying the generation of the events. The sensor nodes are listening (subscribing) for an event. Some other nodes in the network may be listening for same event. The event generation could be any where in the network. The following figure3 shows the usage of Event style in wireless sensor networks. Component1 (com1) is listening for an event. If event occurs then com1 sends data to com2.

The syntax of the event style is event:eventType ? listenerObjA.

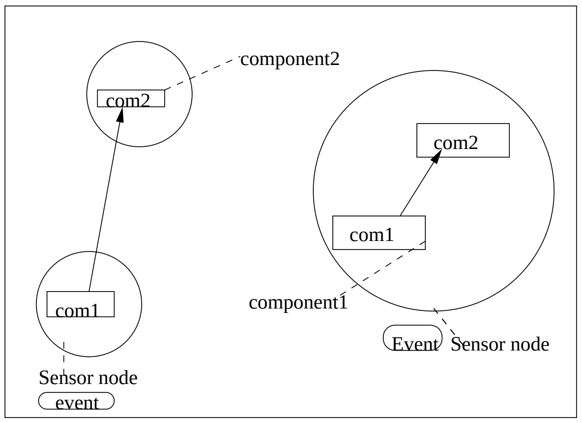

Fig. 3. Example of Event Style

def Component event=Temperature

def Component observer=Observer

run:event ? (Then) observer

Here, event and observer are components. The interaction between components is event style. If event(e.g temperature is greater than 70^0C) occurs then notify the observer. We can use ? or Then as an operator. In the next section we have discussed the usage of styles in sensor network applications.

Pipe Style: The semantics of the style states that the output produced by a component is passed through a pipe. This is received as input from the other end of the pipe. Another component is connected to the pipe at the other end. This style is static in nature. In UNIX the system call pipe(|) also behaves in the above mentioned way. The dynamic nature of the pipe style in this language enables it to be better suited for mobile and wireless sensor network applications.

Pipe Style Over Event Style: The pipe style over event style in this language is an extension of the basic pipe style. This style is binary in nature. It requires two component methods as operands. The left operand(component) is waiting for an event and if the event occurs then send data to the next component (right operand already waiting for receiving data from left operand). This event is captured by the right operand which will be a listener component. This operator can be extended to arbitrarily many components. The key point in this style is that the ends of the pipe are decoupled which makes the ends flexible to move as they are developed and also enables pipe ends to be changed dynamically. This is the main advantage of redefining pipe style over event style. The pipe style becomes an appropriate style for sensor networks as the event could be generated anywhere in the distributed environment. The following figure shows usage of this style in sensor networks. The com1.pa() is subscribed for com2.pb() and com2.pb() is subscribed for com3.pc().

The syntax for the pipe style is run:compA.methodA()| compB.methodB()

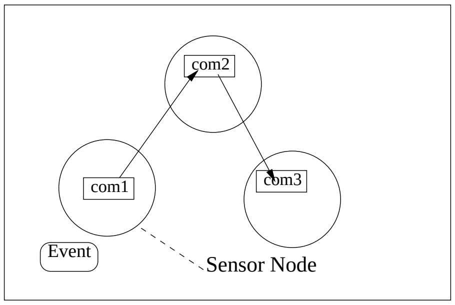

Fig. 4. Example of Pipe Style Over Event Style

Group Communication Style :

The semantics of group communication style states that the person(sensor node) not only communicates within group but also communicates with other groups. Sensor network is divided into logical groups. Each node in a group broadcasts the information not only in that group but also neighbouring groups. Each sensor node may participate in multiple groups. Figure5 shows the usage of group communication style. We used this style in implementing Target tracking application.

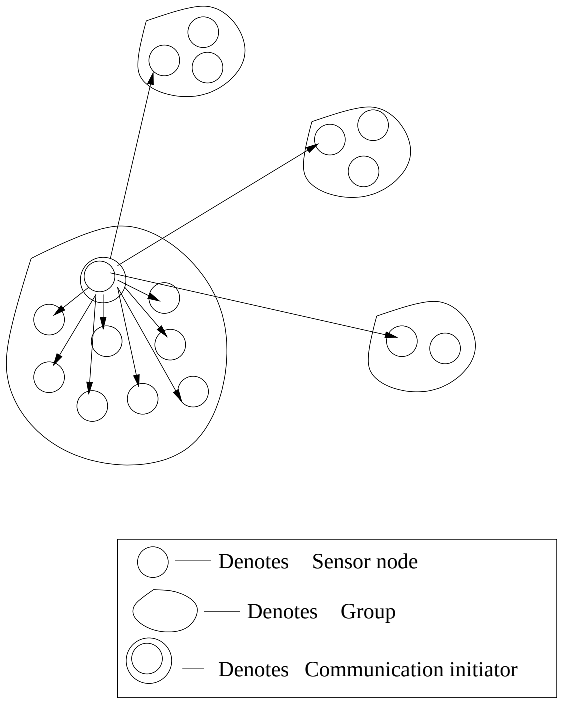

Fig. 5. Example of Group Communication Style

In appendix, we have discussed implementation of styles (Pipe and Event style) in distributed systems.

IV. Applications in Sensor networks

The previous section explained the usage of styles. This section discusses the monitoring and target tracking applications in sensor networks.

Monitoring :

The application discussed here is for controlling chain reaction in nuclear reactors. The sensors monitor the reaction by observing parameters like radiation and temperature. The observer uses data from sensors and maintains the nuclear reactor in a stable state.

This application can be realized in distributed compositional language as follows.

In nuclear reactors, different sensors are used for sensing radiation and temperature. The sensor node senses information and sends to an aggregator node. The aggregator node aggregates the data and sends to a special node called control

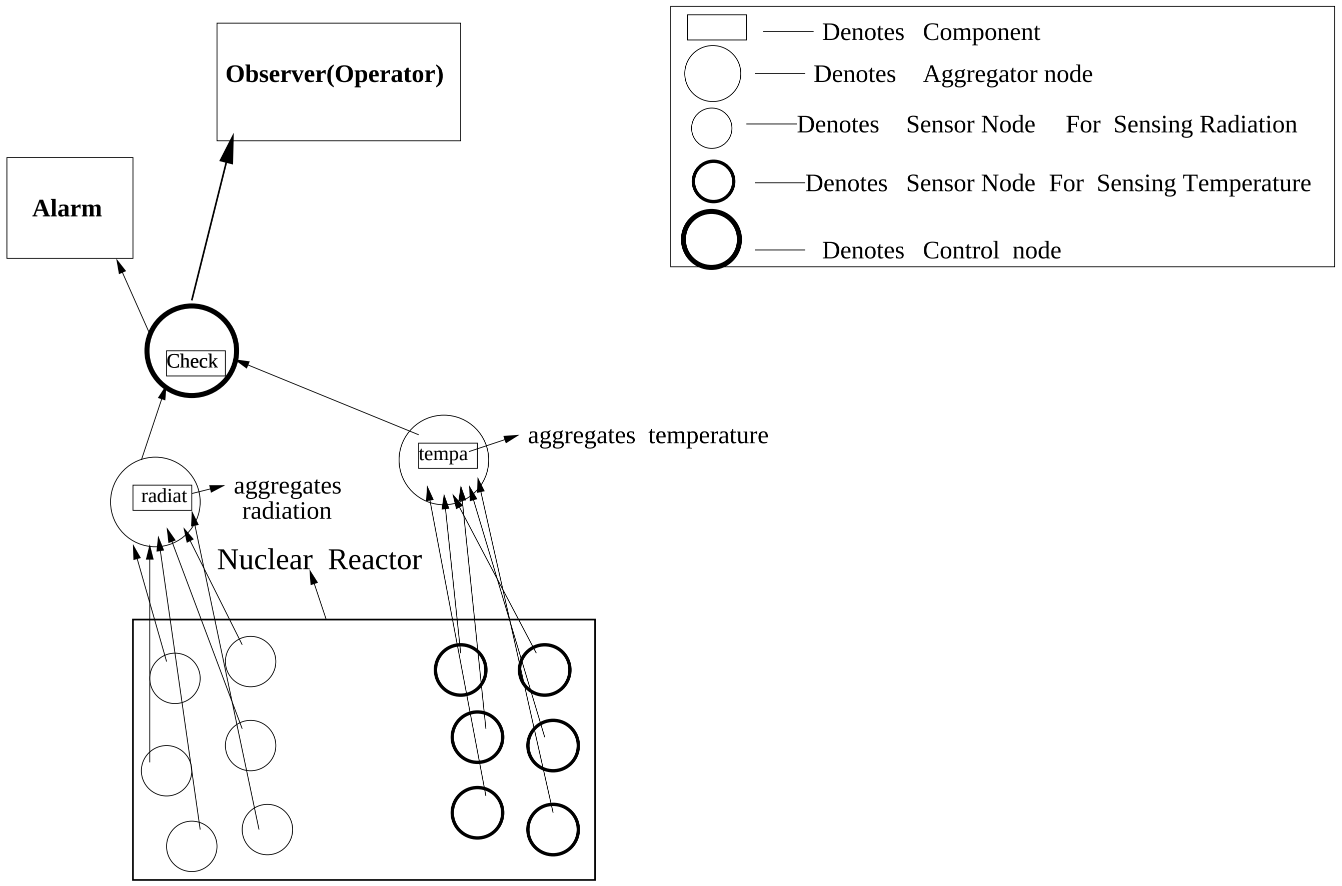

Fig. 6. Monitoring Application

node. The control node sends information to the observer continuously and also checks abnormal conditions like drastic changes in radiation or temperature. If abnormal condition occurs then control node sends information to the observer and as well as to an alarm(actuator).

```
  def  Component com1=Getdata

  def  Component com2=Monitor

run: com1.getdata( )−−>com2.send( )
```

Fig. 7. Script Running at Nuclear Reactor Sensor Nodes

Observer(Operator) can in turn take actions like increasing the coolent(heavy water) flow and reducing reaction rate with the help of control rods(cadmium rods), so that observer can maintain nuclear reactor in a stable state.

The script shown in figure7is running at all sensor nodes in nuclear reactor. In this script, Getdata and Monitor are components. Getdata always takes the data from sensor and passes data to Monitor component. The Monitor component transmits the data to Rdata component in aggregator node. There are two aggregator nodes, one aggregates radiation and another aggregates temperature. The scripts shown in figure 8 and figure 9 contains Rdata and aggregator components. Rdata takes information(radiation or temperature) from Monitor component(which resides in another node) and passes to aggregator component(aggregator component aggregates data). These nodes aggregate information and sends to control node. In figure 10 Control node checks abnormal conditions

```
def  Component  Rdata = Receive_radiation
def  Component  aggregator = aggregate_data_and_send

run : Rdata  Then   aggregator.aggregate(Rdata)
```

Fig. 8. Script Running at Aggregator Node(radiation)

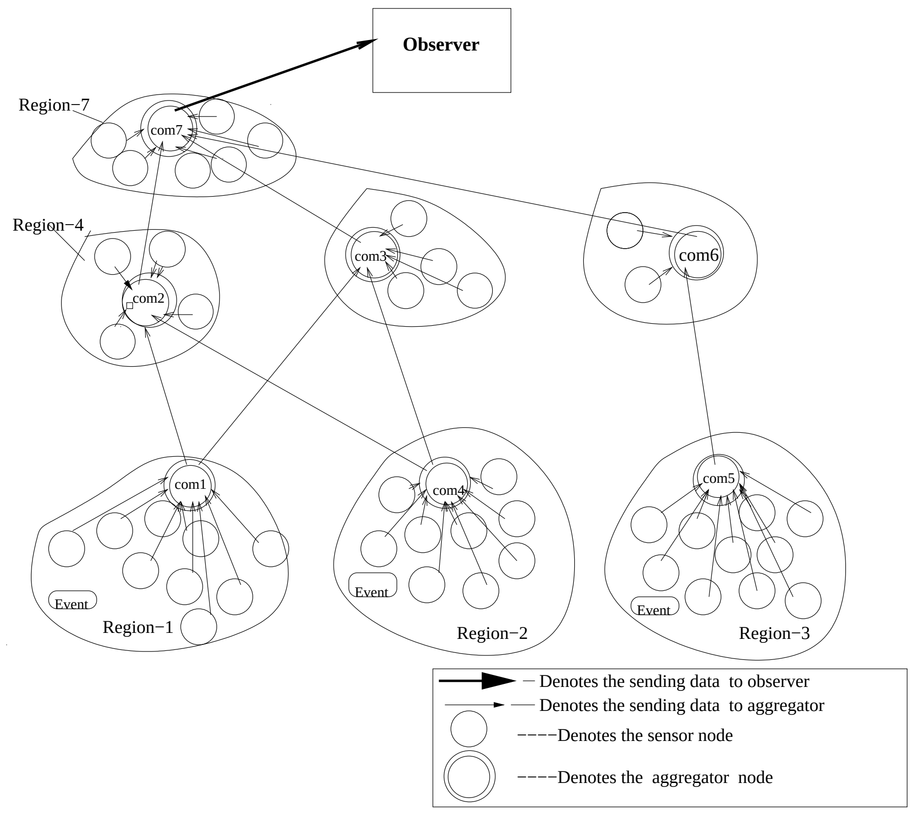
Observer
Region−7
com7
Region−4
com2
com3
com6
com1
com4
com5
Event
Event
Event
Region−1
Region−2
Region−3
— Denotes the sending data to observer
— Denotes the sending data to aggregator
−−−−Denotes the sensor node
−−−−Denotes the aggregator node

Fig. 11. Environmental Monitoring Application

```
def Component Rdata = Receive_temparature
def Component aggregator = aggregate_data_and_send

run : Rdata Then aggregator.aggregate(Rdata)
```

Fig. 9. Script Running at Aggregator Node(temperature)

```
def Component check=Conditions

run: check.conditions()
```

Fig. 10. Script Running at Control Node

by using check component and sends information to the observer(operator).

A simple environmental monitoring application can be realized using pipe style over event style as follows.

The observer monitors temperature of an area using sensors. If any peculiar changes in temperature then the observer receives information from sensor node(near aggregator node).

In this scenario, different components com1, com2$\cdot\cdot\cdot$, com7 are resides in different sensor nodes. These components subscribe to data from other sensor nodes. In all regions(e.g region1, region2, region3 $\cdot\cdot$), the sensor nodes are waiting for a particular event. The event may be abnormal conditions, for example, temperature is greater than 60^0C. If the event occurs then the data is transmitted to the components, which resides in other sensor nodes of the regions, which are near from the observer. In figure 11, the aggregator nodes in region-1 and region-2 send data to region-4, which in turn sends data to region-7. The aggregator node of region-7 sends data to the observer. Figures 12 and 13 show the implementation of environmental monitoring application.

In figure 12, if event(temperature) occurs then Getdata component takes data and passes to Monitor component. The Monitor component sends information to aggregator node in that region. In figure 13 info is a component(receiving information), if aggregator node receives data then info event occurs, so com1(aggregator component) takes data from info component and aggregates data. The aggregator component sends information to another aggregator component, which resides in different region.

Target Tracking :

“Tracking is a canonical problem for sensor networks and

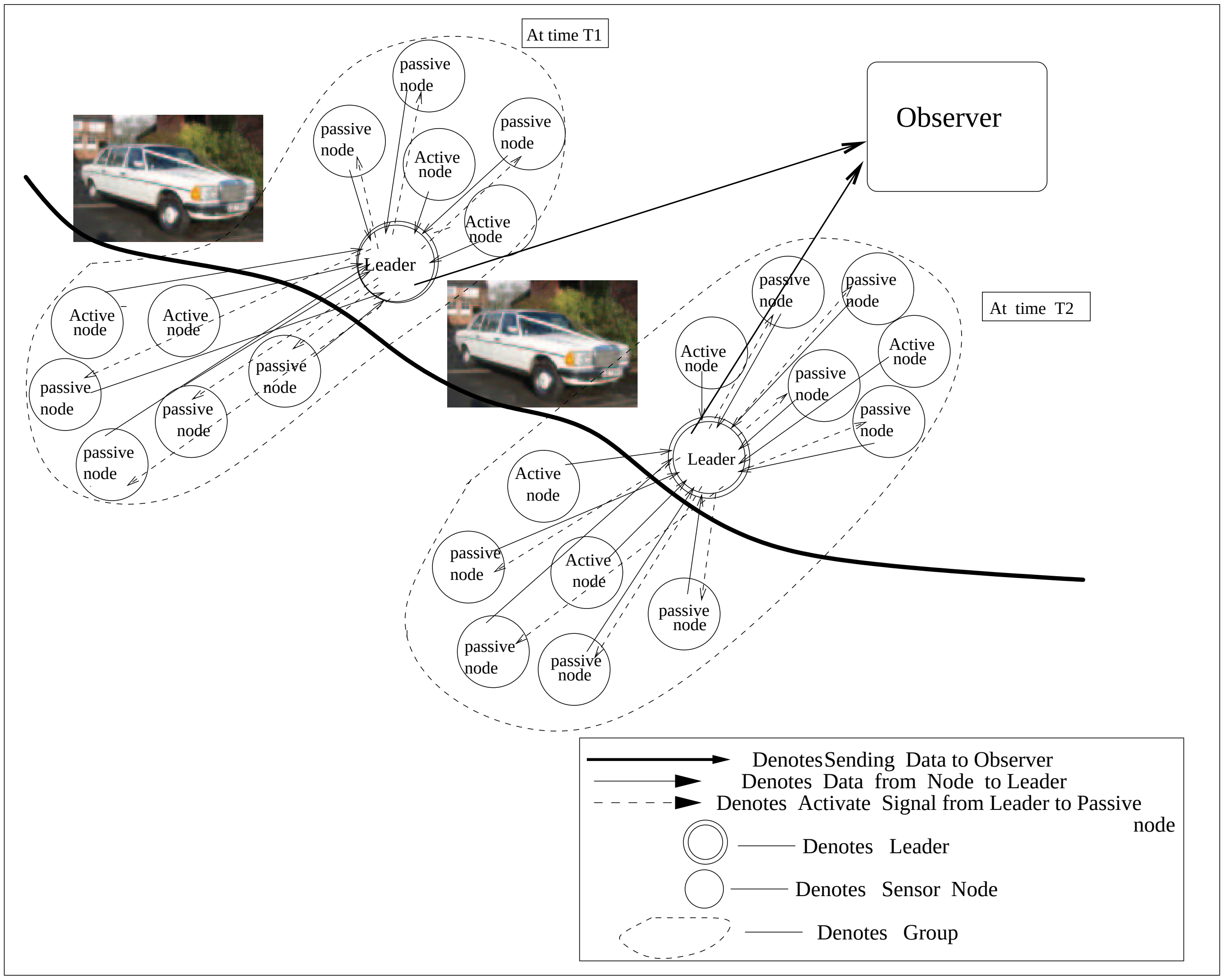

Fig. 14. Implementation of Target Tracking Application

```
      def  Component  event=Temperature
      def  Component com1=Getdata
      def  Component com2=Monitor
run: event Then  com1.getdata()−−>com2.send()
```

Fig. 12. Script Running at Nodes in Regions

```
def  Component info=Receive_data
def  Component com1=aggregator
 run: info ? com1.aggregate(info)
```

Fig. 13. Script Running at Aggregator Node in Regions

essential for many commercial and military applications such as traffic monitoring, facility security, and battlefield situational awareness”[8]. We propose a solution (implementation) using this language. A target is moving in sensor field, we have to estimate target state histories, such as spatial trajectory, on the basis of sensor measurements. Each sensor node provides a local measurement useful in estimating the target state. However, in most cases, only a relatively small subset of sensors contribute significantly to the estimation, due to sensing-range limitations.

```
def  Component  event =Detect_Intruder
def  Component  observer =Observer
def  Component  neighbours =Neighbours
def  Component  neighgroups =Distant_active_ nodes
run : event  Then  observer.notify(event),  neighbours.notify(event),
                                           neighgroups.notify(event)
```

Fig. 15. Script Running at Active Node

```
  def  Component  warnevent=Warning

  def  Component  neighbours=Neighbours

 def  Component  dis_active=Distant_active_ nodes

run: warnevent  Then  neighbours.alert(warnevent)
                          dist_active.alert(warnevent)
```

Fig. 16. Script Running at Active Node

```
     def  Component  event=Detect_Intruder
     def  Component com1=Getdata

     def  Component com2=Leader

run: event Then  com1.getdata()-->com2.leader(event)
```

Fig. 17. Script Running at Sensor Nodes in a Group Except Leader

```
def  Component  Rdata = Receive
def  Component  aggregator = aggregate_data_and_send

run : Rdata  Then   aggregator.aggregate(Rdata)
```

Fig. 18. Script Running at Leader

In Figure 14, the active nodes, which are closer to the intruder(target) forms a logical group. Every group contains a leader. We assume that middleware will take care of formation of a logical group and fault tolerance issues. In order to use energy in an efficient way, we will place most of the nodes in passive state. We keep some of the nodes in active state. Dynamically, one of the active node becomes a leader(An election algorithm is used, to elect a leader).

When an intruder enters the region, the leader(active node), who is closest to the intruder sends information to the observer, neighbour nodes and neighbour groups(distant active nodes).

Now all group members are in active state because leader sends alert information (Activate signal). Within a group, each sensor node retrieves information about the target and sends this to the leader. The leader aggregates data, performs necessary computations to get the most accurate data and sends to the observer. As the target moves, new groups are formed dynamically and the same process continues. When the intruder moves away from the group, then the sensor nodes in that group except the leader are changed from active state to passive state. While the intruder is moving, near distant active node forms a group and elect a leader.

Using the information received from the leader, observer can easily track the intruder(target). Distributed compositional language easily realizes complex applications like target tracking. Figures 15, 16, 17, and 18 show the implementation of this application.

The script shown in figures 15 and 16 are running at all active nodes simultaneously. The first script(shown in fig15) is used for detecting the intruder(target) and the second script(shown in fig16) is used for receiving alert signal from active node, which detects the target(intruder).

In figure 15, Detect_Intruder component is used for detecting the target, Observer component is used for sending information to the observer, Neighbours component is used for sending activate signal to passive nodes, Distant_active_nodes component is used for sending alert signal to closer active nodes so that they can form a group dynamically.

When an intruder enters the region then the leader sends this information to the observer, neighbours and distant active nodes. In figure 16, when the leader sends alert information to distant active nodes then warnevent occurs in distant active nodes. It means the active nodes understand that the target is already in the region and it stops the execution of script shown in figure 15 and it (active node) starts the execution of script shown in figure 18. So the active node sends activate signal to all passive nodes, which are closer to that active node and also sends alert signal to distant active nodes. In figure 17, when an intruder(target) enters the range of a sensor node then the Getdata component takes data from sensor and give it to Leader component. The Leader component sends data to the leader of the group(group formed dynamically). When the target moves away from the sensor node then that node goes to passive state and this is done by Leader component. In figure 18, the leader aggregates data received from all sensor nodes in the group and sends data to the observer. We used pipe style, event style and group communication style in order to realize this application.

The program running (in figure 17) at all sensor nodes uses the pipe over event style because all nodes senses the data and passes to other component through pipe. The scripts(shown in fig15 and fig16) use the group communication style. This is because, it is not only communicating with group members but also communicates with other groups. The script(shown in fig18) uses event style, because it is waiting for data from all sensor nodes(Receive is an event).

V. Related work

The nesC module system is very close the Mesa's [9]and (coincidentally) uses essentially the same terminology: Modules contain executable code, configurations connect components (configurations or modules) by binding their interfaces, whose interface types must match. In case of Distributed Compositional Language, we use styles to connect components instead of configurations. But if we use configuration then developer has to plug the components.

Distributed systems [10], [11], [12], [13]including the CORBA Component Model [14] and Microsoft's COM [15], and Software Engineering [16] often model systems as interacting sets of components. These components are specified by the interfaces they provide or use. However, the focus is very different from Distributed Compositional Language. Components are large-scale (e.g, a database), dynamically loaded and/or linked, and possibly accesses remotely. The component models are more heavy weight than Distributed Compositional Language. So component models (heavy weight) are not used in sensor networks because memory and power are constrained.

The languages commonly used for programming sensor networks do not offer the set of features desired in Distributed Compositional Language (i.e, styles (interaction) kind of programming).

Sensorware[17] is a middleware approach to application development in wireless sensor networks based on the concept of mobile agents and mobile code. Here, the sensor network is tasked by injecting a program into the sensor networks. This program can collect local data, can statefully migrate or copy itself to other nodes and can communicate with such remote copies. Although this approach is most suitable for sensor networks, where nodes once deployed are inaccessible. Complicated applications with a network of different sensor nodes will be difficult to realize, but in Distributed Compositional Language we can easily realize applications consisting of different sensor nodes.

A number of middleware approaches have been devised that treat the sensor network as a distributed database, where users can issue SQL like queries to have the network perform a certain sensing task. Examples of such database like approaches include TinyDB[18], Cougar which are a decentralized approach, where each node has its own query processor that preprocesses and aggregates sensor data on its way from the sensor node to the user. Database approaches though easy to use are limited in their expressiveness and are often not scalable because of their reliability on network wide data structures(e.g, spanning tree of the network, queries sent to all nodes). In contrast many sensing tasks exhibit very local behaviour(e.g target tracking), where only a very few number of nodes are active at any point in time. In Distributed Compositional Language, the scope of applications are not limited.

The SQTL[19](Querying and Tasking in sensor networks) language is used only for homogeneous sensor nodes but Distributed Compositional Language can be used for heterogeneous sensor nodes also.

In Distributed Compositional Language, we provide higher level abstraction than component oriented programming languages (e.g, nesC, J2Me, etc). The main thing is, we are providing styles for composing components. The style specifies how the components are interacting. In other words, how components are plugged, is specified by the style. In case of, nesC, user has to plug the components such that application should be handled.
Currently this language does not support dynamically customizing the behaviour of a sensor node [20] (reprogramming), we will be incorporating this feature shortly.

There exist a few Compositional Languages. Piccola is a π calculus based composition language. Piccola uses forms,agents and channels as its primitive values. The difference between piccola and compositional language is, piccola is not a distributed composition language[6]. So, it does not require middleware, where as our language is a Distributed Composition Language.

VI. Conclusions and Future Work

The primary goal of our language is flexibility. Flexibility means ease of implementing applications. Developing applications in sensor networks is a tedious process due to lack of flexibility in languages (nesC,tiny script,etc) and also resources are constrained. The secondary goal is, capturing interaction between components across different sensor nodes. Distributed Compositional Language is more flexible than nesC. We believe that "Distributed Compositional Language" would play a vital role to implement complex applications like target tracking in wireless sensor networks.

The components and styles in the language can be used to quickly build compact and complex applications. Various styles like pipe, event, dynamic discovery, group communication are different ways of interaction between components. We explained Pipe style, Event style, and Group Communication style by implementing target tracking application . The grammar of the language has been designed, in a flexible way to be used in the script. The compiler and parser to process compositional language scripts have been built. This work aims at simplifying the process of building truly open systems and it is very much useful for collaborative applications. We have discussed language design issues and implementation of styles in the language. We also presented implementation of monitoring and target tracking applications.

Besides that, our middleware will provide functionalities to configure and manage the whole network, where by the scalability and portability of applications increases.

Now we are focusing on applications like ”Fire fighters problem”, ”Intelligent transportation systems”, ”Military domain problems”, etc. From applications, we will develop appropriate interfaces and new interaction styles(if necessary), to develop a wide range of applications. The purpose of our research activities is the development of a frame work, which radically simplifies the development of software for sensor network applications.

REFERENCES

[1] P. Levis and D. Culler, “Mate:a tiny virtual machine for sensor networks,” in *In Proceedings of the tenth International conference on Architectural support for Programming Languages and operating systems ACM press,New York,NY*, pp. 85–95.

[2] P. Levis, “The tinyscript language,” in *A Reference Manual*, 2004.

[3] A. U. Kumar, “Design and implementation of distributed object composition language m.tech thesis,” in *IIT Madras*, 2004.

[4] A. V. Srinivas, D. Janakiram, R. Koti, and A. U. Kumar, “Realizing large scale distributed event style interactions,” in *In the Proceedings of the ECOOP Workshop on communication abstraction for Distributed Systems*, 2004.

[5] O. Nierstrasz and L. Dani, “Component-oriented software technology in Object Oriented Software Composition,” in *Prentice-Hall*, pp. 3–28, 1995.

[6] F. Achermann and O. Nierstrasz, “Applications=components+scripts a tour of piccola,” in *Software Architectures and Component Technology*, pp. 261–292, 2001.

[7] J. Hill, R. Szewczyk, A. W. S. Hollar, D. E. Culler, and K. S. J. Pister, “Sytem architecture directions for networked sensors,” in *In Architectural Support for Programming Languages and Operating Systems*, pp. 93–104, 2000.

[8] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao, “State-centric programming for sensor-actuator network systems,” in *Palo Alto Research Center, published by the IEEE CS and IEEE ConSoc*, 2003.

[9] J. Mitchell, “Mesa language manual technical report,” in *Xerox PARC*, pp. CSL–79–3, 1979.

[10] A. Herbert, “An ansa overview ieee network,” pp. 18–23, 1994.

[11] I. I. S. 10746-3, “Odp reference model architecture,” 1995.

[12] O. M. Group, “Common object request broker architecture,” in *Available at http://www.omg.org*.

[13] S. M. J. Beans

[14] O. M. Group, “Corba component model,” in *Available at http://www omg org*.

[15] “Ole2 programmer’s reference,” in *Volume one Microsoft Press*, 1994.

[16] F. Bachmann, L. Bass, C. Buhrman, S. C. Dorda, F. L. J. R. R. Seacord, and K. Wallnau, “Concepts of component-based software engineering,2nd edition,” in *Technical Report CMU/SEI-2000-TR-008*, May 2000.

[17] A. Boulis, C. Han, and M. B. Srivastava, “Design and implementation of a framework for programmable and efficient sensor networks,” in *MobiSys*, 2003.

[18] S. Madden, J. Hellerstein, and W. Hong, “Tinydb: In-network query processing in tinyos,” in *www.berkeley.edu, Techreport*, 2003.

[19] C. J. C.Srisathapornphat and C.-C. Shen, “Querying and tasking in sensor networks,” in *SPIE’s 14th Annual Intl Symp. Aerospace/Defence Sensing, Simulation, and Control, Orlando.*, 2000.

[20] C. Babu, W. Jaques, and D. Janakiram, “Dynamic customization of pervasive computing environments,” Tech. Rep. IITM–CSE–DOS-04-12, IIT Madras, 2004.

APPENDIX

Pipe Style:

The design of pipe style is shown in Figure 19

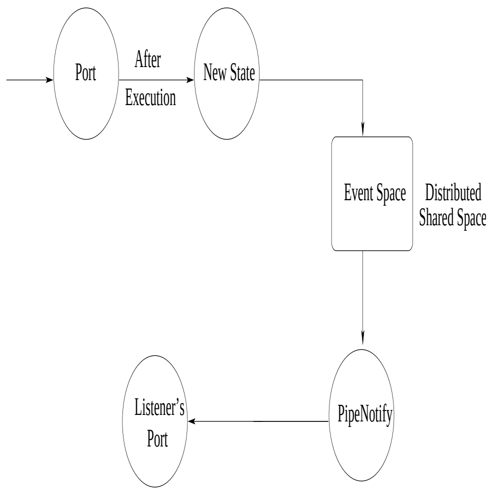

Fig. 19. Design of Pipe Style

The syntax for the pipe style is run:compA.methodA()| compB.methodB()

where compA.methodA() is the method methodA in the component compA. The run time environment of the language invokes methodA and executes the method. Once the method execution is finished, the component compA is loaded onto the event space of the DSM. This generates an event whose listener object is component compB. The method methodB receives the object of compA as a parameter. It is internal to methodB as to how the data variables and methods of compA are used.

Grammar:

PipeStatement ::= RUN PipeExpression;
PipeExpression ::= ID DOT ID PIPE PipeExpression | ID DOT ID;

Syntax: run: com1.portA | com2.portB | com3.portB

Algorithm:

GlueComponent:

1 Call ObjectDataBase to get the objects of symbols com1, com2, com3, etc.,
2 The right hand side operand com2 is placed in DSM making it a listener object.

2.1 The listener object is listening for the event of type com1.(LHS Operand)
2.2 When event occurred then pipeNotification method of listener object is invoked dynamically.
3 The associated method for port portA is invoked.
4 The com1 is made as an event object and placed in event space of the DSM
5 Step 2 is called with com3 as listener object and so on.

Object DataBase:

1. Check the hash table for the object associated with the received symbol and return the object.

Event Style:

The design of event style is shown in figure 20.

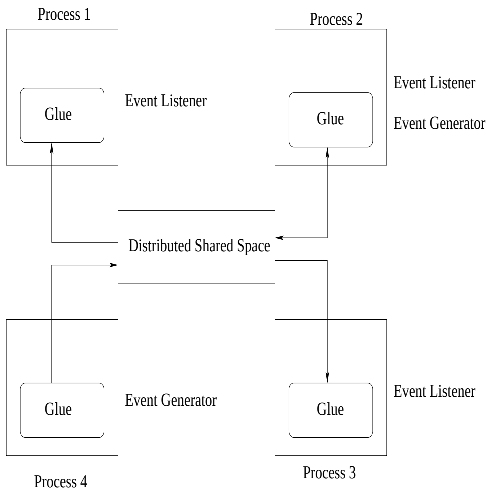

Fig. 20. Design of Event Style

The syntax of the event style is event:eventType ? listenerObjA.

The listener objects listenerObjA, listenerObjB are created and loaded in the DSM (Distributed Shared Memory). The runtime of the language sends notification to the listener objects. When an object of type event Type is loaded onto the DSM through a event generation process then it is considered as an event generation. The asynchronous nature of notification allows the process to work independently of event occurrences. Thus, the processes are not blocked for the events. Using this kind of design, we can realize the publish-subscribe architecture. The event style becomes a very important and key style in the language.

The registration of events process creates a list of listener process in the DSM.When the event is generated by a process, the event type is compared with already registered listeners and the appropriate listeners are notified. Invoking the notify methods of the listeners is at the language level. The notification to the different process across the cluster is done at the DSM level.

This architecture can be used in wide variety of applications like chat, electronic polling, etc.

Grammar:

EventStatement ::= EVENT EventExpression QUESTION EventActionsExpression
EventExpression ::= ID:s1
EventActionsExpression ::= ID DOT ID COMMA EventActionsExpression | ID

Syntax:
event: evenType ? listenerA, listenerB
Algorithm:

GlueComponent:

1 Call ObjectDataBase to get the objects of symbols eventType, listenerA, listenerB.
2 For each listener component a listener object is loaded into the DSM.
3 A list of eventType and their respective listeners are maintained.
4 The listener Objects are waiting for the event in EventHandler.

Object DataBase:

1. Check the hash table for the object associated with the received symbol and return the object.

Event Handler:

1 A request is sent to DSM to check whether the event of type evenType has occurred.
2 If occurred then 2.1
2.1 Request is sent to DSM to get the event Object.
2.2 A new thread is spawned to go to step 1.
2.3 Notify method of listener object is called.