DESAL$^\beta$: A Framework For Implementing Self-stabilizing Embedded Network Applications

Yangyang He[1(✉)], Yuheng Du[1], Shiree Hughes[2], Jiannan Zhai[2], Jason O. Hallstrom[2], and Nigamanth Sridhar[3]

[1] School of Computing, Clemson University, Clemson, USA
{yyhe,yuhengd}@clemson.edu
[2] I-SENSE, Florida Atlantic University, Boca Raton, USA
{shughes2015,jzhai,jhallstrom}@fau.edu
[3] Electrical and Computer Engineering, Cleveland State University, Cleveland, USA
n.sridhar1@csuohio.edu

1 Introduction

The Dynamic Embedded Sensor-Actuator Language (*DESAL*) [2] is a rule-based programming language, without events, interrupts, or hidden control. Nodes have built-in access to their neighbors' state, with automatic node discovery and health monitoring. Applications communicate via shared variables, rather than explicit message passing. Shared variables naturally represent the state of self-stabilizing algorithms. *DESAL* simplifies the construction of self-stabilizing embedded applications by eliminating network programming, while offering significant reliability improvements.

Contributions. This paper presents both incremental and fundamental contributions. First, we present $DESAL^\beta$, a significant improvement of the $DESAL^\alpha$ implementation reported in [1]. $DESAL^\beta$ includes a new, more complete compiler, with new support for C-based types and control flow constructs, as well as a new C/nesC code mixing feature. A comparative performance analysis between $DESAL^\alpha$ and $DESAL^\beta$ is presented. Second, and more fundamentally, we present an in-depth treatment of a self-stabilizing algorithm realized in $DESAL^\beta$ to assess the utility of the paradigm. The analysis centers not only on ease-of-use, but on fault-tolerance and convergence time, post-fault. Prior publications focused on grammar and architectural details.

2 Related Work

Prior work on programming approaches for embedded network systems span two paths. The first is focused on node-level programming. Representative solutions include Contiki [4], MANTIS [5], TinyOS [3], and others. Although the kernel of Contiki is event-driven, preemptive multi-threading is supported through a library. Contiki programs can be loaded dynamically and are C-based. Similarly, MANTIS provides users with a cross-platform, event-driven operating system

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2016
B. Mandler et al. (Eds.): IoT 360° 2015, Part II, LNICST 170, pp. 307–312, 2016.
DOI: 10.1007/978-3-319-47075-7_34

that can be used to load programs dynamically. Programs for MANTIS are written in C, with slight changes to the basic program structure, such as requiring a `start` function instead of a `main` function, as well as other idioms. TinyOS is another platform for wireless sensor networks (WSNs). Programs are written in *nesC*, an event-driven language that derives from C [6].

The second path of prior work focuses on network-level macro programming, which hides the details of individual sensor nodes from programmers. TinyDB [7] and Cougar [8], for instance, abstract a WSN as a relational database and allow the use of declarative SQL-based queries to retrieve data from the network. Kairos [9] provides a shared memory abstraction to access one-hop neighbors and acquire their data. Regiment [10] is a functional language that enables *spatiotemporal* macroprogramming, which hides the direct manipulation of program states from the programmer. It divides a larger network into abstract regions and provides abstractions for querying the state across a region.

$DESAL$ is introduced in [2]. It adopts five fundamental principles: (i) a state-based model of programming, which abandons event-driven logic in favor of state-based logic; (ii) shared variable communication, which enables the sharing of state variables across devices; (iii) a rule-based programming model, in which programs comprise a set of statements, where each statement is a guarded action dependent on a Boolean condition; (iv) dynamic binding, which allows for shared variable communication in the presence of changing wireless connectivity; and (v) synchronized, network-wide action timing. A preliminary implementation of a DESAL compiler is presented in [1]. SELFWISE [11] also supports state-based programming and offers a supporting runtime environment for self-stabilizing algorithms. However, it lacks support for coordinated, distributed actions, which is an important feature in many scenarios.

3 DESAL$^\beta$

$DESAL^\beta$ is a framework for implementing self-stabilizing embedded network applications. It provides a state-based programming language with support for C-based constructs, a runtime platform based on TinyOS and a Java user interface used to monitor and debug applications. We adapted the $DESAL$ runtime to TinyOS 2.1.2, from TinyOS 1.x and replaced the time synchronization module with the Flooding Time Synchronization Protocol (FTSP) [15] to achieve better synchronization performance. $DESAL^\beta$ code is translated to nesC code and compiled with the supporting runtime libraires. $DESAL^\beta$ adapts the runtime design described in [1].

New $DESAL^\beta$ language features include: (i) *C-based structs* to provide flexible data representation; (ii) *nesC/C code mixing* to accommodate situations where it is more convenient to use event-based semantics; (iii) *multi-hop binding* to enable variable sharing across multiple hops, and to simplify algorithms implemented based on the notion of a K-neighborhood; and (iv) *declarative link quality guarantees* to ensure network robustness.

4 Case Study: Spanning Tree

The case study involves creating a routing tree in the network. A key advantage of $DESAL^\beta$ is that complex logic for message exchange is hidden from the programmer. The self-stabilizing algorithm developed by Goddard et al. [18] is:

Rule 1:	**Rule 2:**	**Rule 3:**
$v.ID = 0 \rightarrow v.distance = 0 \wedge v.parent = v;$	$v.ID \neq 0 \wedge \exists u \in N(v) : (u.distance = minD) \rightarrow v.distance = minD + 1 \wedge v.parent = u \wedge v.parentAlive = true;$	$v.ID \neq 0 \wedge v.parentAlive = false \rightarrow v.distance = \infty \wedge v.parent = v$

where *v.distance* represents the distance of node v from the root of the tree, *v.parent* points to the parent node of v, and *minD* denotes the current shortest distance to the root, among all neighbors of a non-root node v. An ID value of zero is used to represent the root node, while non-zero values represent non-root nodes. *v.parentAlive* indicates whether a node's parent is healthy. Rule 1 is responsible for declaring the root node of the tree. Rule 2 is responsible for searching for a parent. Rule 3 is responsible for recovering from a parent fault.

We translate the above rules to the $DESAL^\beta$ code shown in Listing 1. Line 2 shows that each node maintains a shared variable `distance` to represent its distance from the root. Initially, each node's `distance` is set to 255, indicating a disconnect from the tree. On line 3, local variable `parent` is similarly initialized to the host node's `ID`. Neighbors' distance information can be read from the multi-binding `nDistance`. The first subcomponent, on lines 9–23, implements rules 1 and 2. Every 3 s, the `foreach` loop on lines 11–17 finds the neighbor offering the shortest distance to the root. The `ID` of this node is acquired by `src()` and used to update `parent`. The `parentAlive` flag is set to *true*, and the host node's `distance` is set to 1 hop greater than the parent's distance, `minD`. The root node's `distance` is set to 0, as shown on lines 22–23. Rule 3 is implemented on lines 26–41. The `src()` function is used to (implicitly) check whether the parent is still active. It returns the source ID of a binding. If it returns the parent ID in the `foreach` loop, it implies that the parent is still reachable, and `parentAlive` is set to *true*. When a non-root node detects a failed parent, it resets its distance (255) and gets ready to rejoin the tree, as shown on lines 9–19.

```
1  component spanningTree
2    shared uint16 distance = 255
3    unshared uint16 parent = ID
4    unshared uint16 minD = 254
5    unshared bool parentAlive = false
6    binding uint16 nDistance <-
           *.spanningTree.distance[20]
7
8    // non-root node updates its parent
9    every 3\,s after 0s
10     (ID!=0 && parentAlive == false):
11       foreach d in nDistance {
12         if(d<minD){
13           minD = d
14           parent = src(d)
15           parentAlive = true
16         }
17       }
18       distance = minD+1
19       $Leds(distance)
20     []
21     // set root node's distance
22     ID == 0:
23       distance = 0
24
25     // check parent, recover if parent down
26     every 30\,s after 0s
27       (ID !=0 && parentAlive == true):
28         parentAlive = false
29         foreach d in nDistance {
30           // determine if parent is alive
31           if(parent == src(d)){
32             parentAlive = true
33           }
34         }
35     // if parent down, reset
36     if(parentAlive == false){
37       minD = 254
38       distance = 255
39       parent = ID
40       $Leds(distance)
41     }
```

Listing 1. Spanning Tree

The $DESAL^{\beta}$ source code for the spanning tree algorithm is realized in only 35 lines of non-whitespace code. The conciseness of $DESAL^{\beta}$ makes the implementation of each rule a natural process and requires no understanding of the underlying nesC facilities.

To validate the application, we performed a simulation in Cooja with 35 randomly located nodes, each running the TinyOS image created from the $DESAL^{\beta}$ application. We specified a given node (node 35) to be the root node. Upon stabilization, the network organizes itself into a tree, and the parent of a given node (node 31) is node 9. Since `distance` is the only state variable shared within the network, we inject a parent fault by setting the distance of node 9 to 255. We see that after the next round of communication, node 31 has been accepted by node 12, its other neighbor, as its new child. This fault is corrected in 9.2 s, which is approximately the convergence time of the algorithm, as discussed later.

5 Evaluation

5.1 Space Overhead

We compare the space overhead for the spanning tree application using $DESAL^{\alpha}$ and $DESAL^{\beta}$. Nescc 1.3.4 [13] is used to collect the memory usage data. The $DESAL^{\alpha}$ program uses 17,516 bytes of ROM, and the $DESAL^{\beta}$ program uses 24,476 bytes. The difference in ROM usage is mainly due to the introduction of a more sophisticated time synchronization module. $DESAL^{\beta}$ uses TinyOS 2.x's TimeSync library, which introduces 5,128 bytes more ROM overhead than the TinyOS 1.x module used previously. ROM usage is also increased due to the new communication stack in TinyOS 2.x. Test results show that TinyOS 2.x introduces 2,768 bytes of ROM overhead when two parameterized `AMSend` interfaces are used. Since TinyOS 2.x uses more precise RAM allocation for timers [23], the RAM usage of the $DESAL^{\beta}$ program is smaller, at 214 bytes. Note that RAM is much more scarce than ROM on most embedded network platforms. Consequently, the decrease in RAM usage by $DESAL^{\beta}$, even at the expense of increased ROM usage, is a significant efficiency improvement.

5.2 Convergence Time

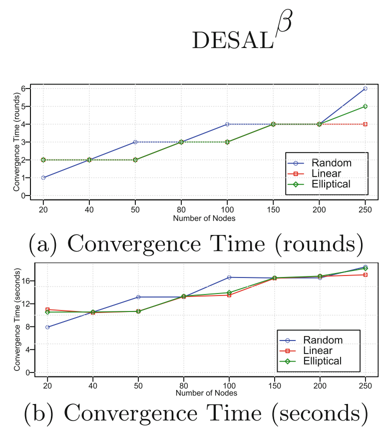
When a fault occurs in a self-stabilizing network, the system eventually converges to a legitimate state. A key performance measure for self-stabilizing algorithms is the time taken for convergence. Convergence is defined by a global predicate, which can often be expressed as a conjunction of local predicates. We measure the convergence time of the $DESAL^{\beta}$ application by tracking the local convergence time of each node. Network convergence time is expressed as the maximum local convergence time in the network.

To measure convergence time, we used the NESTBed [14], which consists of 80 Tmote Sky nodes arranged in a grid topology, as well as the Cooja simulator. We first use the testbed to test networks consisting of 20, 40, 50, and 80 nodes.

We then use Cooja to test networks of 100, 150, 200, and 250 nodes. In Cooja, each algorithm is simulated on 3 different topologies in a 100 m*100 m area. A random topology places the nodes randomly, a grid topology places the nodes evenly over the region in a matrix format, and an elliptical topology places the nodes in a circle over the region. Each simulation assumes a Unit Disc Graph Medium (UDGM), with a 100 % reception rate inside the disc, and 0 % outside. Collisions occur if two nodes transmit concurrently.

For the Spanning Tree algorithm, the local predicate is $(n.parent \neq n) \wedge (n.Distance = n.parent.Distance + 1)$. Figure 1 shows the convergence time of the algorithm. Figure 1a shows convergence time in terms of rounds, as a function of network size. Figure 1b shows convergence time in terms of wall clock time, again as a function of network size. Both metrics grow as expected with increased size.

From Fig. 1, we observe that the elliptical topology has the lowest convergence time of the three topologies. This is because this topology results in the least number of message collisions compared to the other topologies. Also, as network size increases, it becomes more difficult for the algorithm to stabilize. This is because our experiments are performed within a limited area (100 m*100 m); the probability of message collision significantly increases as network density increases. We also tried to enlarge the network to more than 300 nodes. However, the simulation became too slow due to the RAM consumption caused by the increase in Java threads in the Cooja application.



(a) Convergence Time (rounds)

(b) Convergence Time (seconds)

Fig. 1. Spanning tree

6 Conclusions

In this paper, we presented *DESAL*$^{\beta}$, a framework for implementing self-stabilizing embedded network applications. A spanning tree algorithm was used to demonstrate that using *DESAL*$^{\beta}$ to develop self-stabilizing embedded network applications is significantly easier than using event-based programing languages, such as nesC. However, when it is more convenient to use event-based semantics, applications can also be written using a mixture of *DESAL*$^{\beta}$ and nesC/C. Experimental results show that the space overhead of *DESAL*$^{\beta}$ is acceptable and convergence time is low.

Acknowledgment. This work is supported by the NSF through award CNS-0746632.

References

1. Dalton, A.R., et al.: *Desal*$^{\alpha}$: an implementation of the dynamic embedded sensor-actuator language. In: Proceedings of the ICCCN 2008, vol. 8 (2008)
2. Arora, A., et al.: A state-based language for sensor-actuator networks. ACM SIGBED Rev. **4**(3), 25–30 (2007)
3. Hill, J., et al.: System architecture directions for networked sensors. In: Proceedings of the ASPLOS IX, pp. 93–104 (2000)
4. Dunkels, A., et al.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: Proceedings of the LCN 2004, pp. 455–462, November 2004
5. Bhatti, S., et al.: Mantis OS: an embedded multithreaded operating system for wireless micro sensor platforms. Mob. Netw. Appl. **10**(4), 563–579 (2005)
6. Gay, D., et al.: The nesC language: a holistic approach to networked embedded systems. In: Proceedings of the PLDI 2003, pp. 1–11 (2003)
7. Madden, S.R., et al.: TinyDB: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst. **30**(1), 122–173 (2005)
8. Yao, Y., Gehrke, J.: The cougar approach to in-network query processing in sensor networks. SIGMOD Rec. **31**(3), 9–18 (2002)
9. Gummadi, R., Gnawali, O., Govindan, R.: Macro-programming wireless sensor networks using *Kairos*. In: Prasanna, V.K., Iyengar, S.S., Spirakis, P.G., Welsh, M. (eds.) DCOSS 2005. LNCS, vol. 3560, pp. 126–140. Springer, Heidelberg (2005)
10. Newton, R., et al.: The regiment macroprogramming system. In: Proceedings of the IPSN 2007, pp. 489–498. ACM, New York (2007)
11. Weyer, C., Turau, V.: SelfWISE: a framework for developing self-stabilizing algorithms. In: David, K., Geihs, K. (eds.) Kommunikation in Verteilten Systemen (KiVS), pp. 67–78. Springer, Heidelberg (2009)
12. Osterlind, F., et al.: Cross-level sensor network simulation with COOJA. In: Proceedings of the LCN 2006, pp. 641–648, November 2006
13. NESCC. linux.die.net/man/1/nescc
14. Dalton, A., et al.: A testbed for visualizing sensornet behavior. In: Proceedings of the ICCCN 2008, pp. 1–7, August 2008
15. Maróti, M., et al.: The flooding time synchronization protocol. In: Proceedings of the SenSys 2004, pp. 39–49. ACM (2004)
16. McPeak, S., Necula, G.C.: Elkhound: a fast, practical GLR parser generator. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 73–88. Springer, Heidelberg (2004)
17. Hedetniemi, S.M., et al.: Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. Comput. Math. Appl. **46**(5–6), 805–811 (2003)
18. Goddard, W., et al.: Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In: Proceedings of the IPDPS 2003, p. 14. IEEE (2003)
19. Mahafzah, M.H.: An efficient graph-coloring algorithm for processor allocation. Int. J. Comput. Inf. Technol. **02**(1) (2013)
20. Johnson, D.S., Garey, M.R.: Computers and Intractability. Freeman, New York (1979)
21. Hedetniemi, S.T., et al.: Linear time self-stabilizaing colorings. Inf. Process. Lett. **87**, 251–255 (2003)
22. Moteiv. Tmote sky (2005). http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf
23. Levis, P.: Experiences from a decade of tinyos development. In: Proceedings of the OSDI 2012, pp. 207–220. USENIX, Berkeley (2012)