Integrated development environment for debugging policy-based applications in wireless sensor networks

Nidal Qwasmi, Daniel Smullen*, Ramiro Liscano

University of Ontario Institute of Technology, 2000 Simcoe Street North, Oshawa, Canada, L1H 7K4

Abstract

Autonomic and customized behavior in Wireless Sensor Networks (WSN) can be facilitated through the support of policy-based programming. Currently, a facility for policy debugging and execution trace does not exist. The state of the art assumes that developers will use other methods such as static analysis, batch scripts, and the emulators/simulators to achieve debugging and testing through unsustainably complex gauntlets of manual tests. In this paper, a new approach for controlling the policy-based WSN application simulation is proposed by creating a Policy Integrated Development Environment (Policy IDE) to work interactively with the TOSSIM simulation software using a packet communication mechanism. The benefits of this approach are truly interactive simulation, granular unit testing, interactive debugging, execution trace to validate application logic, and a simplified debugging process for applications that involve policy programming.

© 2013 The Authors. Published by Elsevier B.V. Open access under CC BY-NC-ND license.
Selection and peer-review under responsibility of Elhadi M. Shakshuki

Keywords: Policy-based programming; Integrated Development Environment; Wireless Sensor Network simulation; TOSSIM; Application debugging; tinyOS;

1. Introduction

Machine to machine communications can involve a complex series of interactions that are designed to be autonomous. This can include medical specific applications such as human body sensor networks, among countless other domain-specific usages toward ubiquitous computing and the Internet of Things. Each of these applications requires flexible control logic for regulation of battery usage, signal transmission, and interactions with other onboard systems such as sensors and data processing routines. For sensor nodes with very specific functionality this logic is hard coded by the developer using a specific language. When devices belong to an organization, or must be tailored for the environment, the use of policy languages is better suited.

* Corresponding author. Tel.: +1-647 678 9449.
E-mail address:daniel.smullen@uoit.net.
A policy is a constraint on the system behaviors, which can also be expressed using natural languages or mathematical notation. However, neither of these two extremes is an ideal approach to be used by computer systems [1]. Policy-based systems try to strike a balance between the two extremes by creating a policy language that can fulfill the requirements of the targeted system. Hence, policy languages are declarative and not procedural, which means they express the constraints on the system behaviors but do not specify how these constraints ought to be enforced. This relationship is depicted in Fig. 1. Actions are tied to events, which are mediated through control logic (the policies themselves). Policy systems are typical used for enterprise/network management and access control. The two most typical types are Obligations (O) and Permissions (P). An O policy is traditionally triggered by an event and will specify an action to be performed. A P policy is used to allow or deny a request to a resource on a device.

As an example of the use of policies in sensor networks, we briefly describe the body sensor network example originally in Keoh et al.[2]. In their example there is a sensor measuring the glucose level of a patient. When the level of glucose is greater than 125 and the current time is later than 20:00, the policy will trigger an action to increase the dosage on an insulin pump by 10%. Generally these policies are specified using a GUI which are missing most of the powerful features of IDEs which incorporate tools for debugging, building, and execution of code in tool-chains with varying degrees of automation [3]. The declarative nature of policy languages abstracts the details of a programming language focusing only on the logic required to make decisions. This makes it simpler to develop and customize applications but the consequence is a loss of flow in the application logic. Developing applications using policy languages requires IDEs that can support the emulation and/or simulation of the execution of the policies.

In this paper, we present an IDE for the development of policies for sensor networks. The IDE leverages the Finger policy system that was developed for WSN based on tinyOS [4].

2. Related work

Policy systems have seen widespread usage in aspects of network management, and are considered to be vital to the management of distributed systems [5], [6]. Their usage in WSN stems from this utility, and from further efficacy in software engineering for that platform. Autonomous policy-based systems for WSN and embedded systems are not a novel concept. As shown in [7] and [2] the concept of Self-Managed Cell architecture is used, which lead towards the development of policy systems in WSN for the human body. Finger [4] is a middleware policy-based system designed for body sensor networks. The utility of policy-based management goes beyond this domain, however, in that it allows autonomous behavior. This is possible with other reprogramming methodologies, but policy systems facilitate control logic to be loaded dynamically, enabled, or disabled without shutting down nodes. Finger by itself does not have any inclusive means of creating policies and simulating their evaluation. This was explored by Starfish [8], which builds on the Finger2 policy system. Starfish provides a series of predefined policies and a client-side policy editor. Both Finger and Starfish can interact with several different WSN motes or be used with TOSSIM, a tinyOS emulator and hardware simulator [9]. Regardless of whether a simulator or sensor motes are used, the lack of interactive feedback or visible test artifacts (aside from those added
with instrumentation) means that it is not possible to evaluate the functionality of policy systems interactively. Underlying methods wired into tinyOS also cannot be evaluated using these systems alone.

Other approach attempts at introspection into application behavior and code execution have used static analysis. One approach used state machine derivation to mimic the dynamic behavior of a tinyOS binary [10]. This approach is useful when generating simplistic automata to aid in understanding and error detection within simple programs. This does not scale well into increasingly complex applications, and may become confusing for software with complex logic. While static analysis toward the creation of automata is useful for detecting bugs in some circumstances, it does not have the versatility that traditional interactive analysis and debugging methods have. It cannot be used to directly infer program behavior or characteristics. That is to say, while a state machine allows one to ‘step’ through the states of the program from a theoretical standpoint, it does not necessarily reflect the underlying lines of code, and the intent of the programmer may become lost. This is prominent when using policy systems, which can complicate the underlying functionality of the system from a state-level perspective even in the most basic use cases.

A different approach utilizes injection of instrumentation into program code. This approach has taken various styles; one way is to modify the nesC compiler to inject code, which collects data at various points throughout the program to gather useful debugging data for offline analysis after a run. A visualization toolkit lies at the core of a system which adds probes and records events thrown by the operating system [11]. These approaches are useful in that they provide a means of introspection into the software, but none of these allows this process to occur interactively. They must run in batch-processing jobs. If these instrumentation approaches are used in dynamic analysis with policy-based systems, tests must be predefined and the results of these probes are not useful until execution is finished.

Other approaches to surmounting the reprogramming challenge with tinyOS worked toward building a new operating system. This is designed around a shared kernel and program modules that are easily interchangeable [12]. While this can, efficiently tackle the same problem area which policy systems are useful for, they still fall victim to the same inefficiencies as systems like Deluge [13]. The main failing is that reprogramming sensor motes requires an entirely new system binary to be transmitted over the network in each instance that a change in logic is required. This has significant bandwidth and power consumption overhead, and additionally requires rebooting the node [14]. No system yet exists which allows for the efficient and autonomous reprogramming of policy-based systems in combination with effective instrumentation. This system must support interactive dynamic analysis for the policies to be evaluated, but this problem has not yet been addressed.

3. Showcasing system functionality with a basic domain-specific usage

In this section we present a high level perspective of the use of the policy IDE through a simple example. This generic example has many of the same features as more complex examples. Similar to Starfish, our policy IDE leverages the Finger policy framework with the added ability to execute and test these policies because our policy IDE is integrated with TOSSIM (an emulator, network simulator for tinyOS). In our example a tinyOS nesC, timer and LED component were ‘wired’\(^1\) into the Finger policy system. Wiring these components allows for the sensor mote’s timer and LED to be used and connected with policies. Both components are instrumented with debugging statements that print to the console when events associated with them occur. For example, rather than activating the LED component every arbitrary time interval, a sensing component can be activated instead, effectively

\(^1\) Wiring is a nesC programming concept that associates a reference to a component in one component to the actual implementation of the component.
controlling the operating frequency of the sensor through a policy. The ability to evaluate whether the system is functioning conformant to requirements is critical, and this can be achieved using the debugging tool.

Fig. 2 illustrates the architecture of our solution. Here, TOSSIM is connected to the Policy IDE through a communications channel that is created via packet injection. A Java message passing interface allows for the creation of packets. They contain the instructions for Finger to create, delete, enable, or disable policies. They can also invoke wired events. This interface is interacted with using a Python GTK based GUI, which has supporting backend classes for the construction of packet fields required by Finger. Messages are sent via Java to TOSSIM, through its packet handler to the destination sensor mote.

![Policy IDE Architectural Diagram](image)

This is similar to the Starfish policy GUI but includes additional functionality more typical to those found in IDEs. This secondary contribution addresses the overarching requirement of a mechanism to interact with the simulation easily. Fig. 3 is a screen dump of the policy IDE showing the tab used to create and deploy a policy onto a sensor node. This example toggles activation of the LED when the timer reaches a value set to three (which represents the number of times the timer has been fired).

This particular example reflects the Finger policy format. Message Type is a pre-defined tinyOS message type for the loading of policies onto the motes. In Finger, the policy message type (message type=40) supports source and target addressing for the mote for transferring policies between motes. Such simulation use cases may also require multiple virtual sensor motes. Every implemented policy must have a unique Policy ID. In Finger, there are several pre-defined predicates like equals, less than, greater than, etc. that can be applied to the arguments that are included with the event that triggers the policy. Currently the predicate is applied only to a single argument, which is the first argument in the argument list. In the example in Fig. 3 there is a counter argument included with the timer event that has to be equal to 3 before the policy fires. The action ID defines a tinyOS task that is executed when the policy fires and its associated action arguments. In this example, the argument value “2” is used to specify LED 2.

Other interface tabs allow for interactivity with several other aspects of the simulation and the underlying policy programming. The ‘Simulation Variables’ tab allows developers to adjust the number of simulated sensor motes in the simulation environment. The ‘Remove Policy’ tab allows policies which have been loaded onto virtual motes to be removed. ‘Enable’ and ‘Disable Policy’ allows for loaded policies to be arbitrarily disabled (which prevents their evaluation from occurring), or enabled. ‘Trigger Event’ allows developers to invoke individual events which are governed by policies. These events must
be wired into actions for anything to occur, but in instances where instrumentation has been added to the event code these events are visible even if they have no resultant action. Events can be anything from the device battery reaching a certain threshold, to a LED switching on or off.

Fig. 3. Loading a policy (from the Policy IDE user interface)

The engineering design process for the tinyOS platform requires developers to plan which operating system components, and hardware specific components must be utilized. This can be achieved through object model diagrams that capture the interaction of these objects and their interconnections. At the low-level implementation, interconnection between program modules is achieved by ‘wiring’ components together. This permits the methods and events provided by a component to be invoked and handled, respectively. The Finger policy platform is assumed to be wired into the design when developing applications in these instances. With this done, policy development can occur.

In our simple example, the timer and LED components were used. However, complex examples for different applications could use any number of additional components. As with developing any tinyOS application to support additional components, new components must be wired in using nesC code. After this has been done, and appropriate instrumentation added, the events provided by these components can all be administrated using policies developed in the debugging environment. The events and components must be wired for their code to be executed at all. Moreover, in order for the debugging tool to facilitate the execution of this code, the events must be added to the enumerated list of events supported within the Policy IDE. Once they are added, events can be invoked and policies can be developed for them. Then, the policies themselves can be administrated. This process is as follows:

- Define the events and write the code in nesC that supports them. Wire in your events into the Finger policy engine.
- Define the actions that you want associated with the events, and then write the code that supports them. Wire them into the policy engine.
- Write the policy itself. Associate the action with the event and define the parameters for the policy’s execution.

The strength of Policy IDE is the ability to observe the system state during the execution of actions connected to policies. In order to do this, instrumentation to the code must be added. One of the main contributions of Policy IDE is the addition of comprehensive debugging statements to the Finger platform. As a result, all policy interactions are now visible. For application, specific debugging the burden is put on the developer to implement additional debugging statements to new actions that have been added to support new application specific functionality. This may include code for sensing devices,
Fig. 4 (a) is the log of the output statements associated with the loading of a policy to a mote. These output statements have been added to the actual Finger components to help determine if the Policy IDE is functioning properly. Highlighted in red is an actual output statement verifying the delivery of the policy load message (AMPacket = 40). This particular output is an internal message from the debugging environment itself, and does not necessarily reflect the response or the resultant output coming from the simulated sensor nodes inside TOSSIM. Fig. 4 (b) shows the output produced from the embedded instrumentation of the timer (highlighted in blue) as well as the instrumentation from the Finger platform (highlighted in red). Statements prefixed with DEBUG are messages coming from the simulation instrumentation itself. The address of the sensor mote relaying the message is enumerated within the parenthesis. Viewing this output demonstrates the utility of the policy development environment. Here, a policy can easily be developed and implemented as in previous systems [8], but this functionality is further extended by introducing the ability to view the results as well. By interacting with the simulation further, the results of implementing and invoking policies becomes visible interactively and the contribution of the Policy IDE becomes evident. The result of the evaluation of policies is a reprogrammable system that can activate a wired component at an arbitrary time interval, and artifacts of the execution are visible in the Policy IDE debugging log interactively. After execution, the simulation remains paused until the next interaction occurs. In contrast with past methodologies, the ability to modify the implemented policies and disable/enable them on the fly is now available. Previous approaches used automated batch scripts to execute test suites. These interacted with TOSSIM, but without the ability to use premade control-packets, there was no capacity to engage simulation functions interactively or arbitrarily trigger the execution of code within the simulation until now. Here, parameters are constructed into the packet through fields interacted with by the user.

Moreover, manual triggering of events through the interactive debugger enables developers to evaluate the results of a policy that has not yet been deployed to mote. This allows individual unit tests to be invoked on events that have been implemented but not deployed. It also enables developers to isolate
policies from events. Isolation allows debugging of problematic sequences of events, which may be invoked several times indiscriminately through policies. Additionally, policies may or may not be chained to one another (that is to say, via events that trigger other policies). This is a valuable tool for performing the equivalent of unit testing on policies. As was previously pointed out, the declarative nature of policies results in the potential loss of intended flow of the firing of the policies. Thus it is crucial to be able to perform this unit testing on the policies. The inferior alternative is to attempt manual isolation of the behavior of one policy in the center of a series of policies, which are evaluated simultaneously, unassisted and antecedent. The level of granularity can be controlled by enabling or disabling as many policies as necessary to fine-tune the debugging process.

When a policy is interacted with, the simulation executes for approximately 4ms, and then executes the next 250 events regardless of how long they take. By design, TOSSIM is always paused and only executes when the simulation execution is directly invoked. For this reason, the amount of time and subsequent execution of events was chosen in order to ensure that the simulation will evaluate to the point that it has reached a stable state after the introduction of a change to the policy programming. This important advancement allows for the evaluation of the simulation to be interactive.

4. A Guide to the Effective Debugging of Policy Languages

Debugging is one of the most difficult tasks in software development. It is time consuming and tedious. Debugging in tinyOS is even more difficult due to the lack of sufficient debugging tools, which can be integrated with an IDE. The presented approach is to use debugging statements (denoted as “DEBUG”) as an instrument to communicate the program execution status to the tool’s GUI, which is output to the screen through the Debug Log. Consequently, very important questions emerge like; where does one locate the debugging statements? What should be output in these statements? Unfortunately, there is no fast and hard rule to answer these questions because their answers can vary depending on the application itself, and the developer’s preferences.

Guidelines were developed to answer where one locates the debugging statements:

- A debug statement should be added at the beginning of each program module or component to ensure that the code entry point and exit point are captured.
- A debug statement should be added after each triggering of an event, such as receiving a packet, sending a packet, or firing a timer.
- A debug statement should be added after the completion of each event using a split-control process, such as completing a transmission task.
- A debug statement should be added after a critical variable changes value. In our case, this is particularly important when policy variables change or the policy repository contents change.
- A debug statement should be added when a critical decision point is reached. In our example, important decision points include checking the policy repository capacity, policy evaluation, and the results of an action.

To answer the question “What should be the output in these statements?” is heavily dependent on the developer’s judgment and level of expertise. A general rule of thumb is to show a clear and meaningful message. Some guidelines were developed to this effect:

- Use clear and brief descriptions in debugging messages, which detail what has occurred.
- Include module and component names in debugging messages.
- Include variable names and values, where applicable, in debugging messages.
- Use the correct type handling and presentation for variable values, such as hexadecimal presentation for numerical values.
5. Conclusion and future work

While policies enhance the autonomous behavior of WSN systems, they add to the complexity of the debugging process. We have developed a new approach to controlling the TOSSIM WSN simulator in conjunction with a graphical user interface and packet injection mechanism to surmount these obstacles. As a result, interactive simulations, granular unit testing, interactive debugging, and execution trace are possible for policy centered applications. This augments and streamlines the policy development process in particular by enabling developers to develop, deploy, and test policies before they are used in production environments and on hardware sensor motes.

Policy IDE requires the TOSSIM simulator to provide underlying functionality which is unavailable when working with hardware sensor motes directly. A packet injection mechanism similar to that which TOSSIM provides would need to be created and integrated into the system in order to communicate with the underlying policy platform code. Sensor motes do not have the capability of accepting commands injected into them directly without some form of intermediary interface. One possible option is to program a single sensor mote to relay policy packets to other devices. This base station would have a direct connection to a desktop PC so that it could be controlled and used as an intermediary. Further, a mechanism for collecting the data and debugging messages generated by the embedded instrumentation would be required. TOSSIM allows for this data to be redirected to the console or a log file. This behavior is used by Policy IDE to perform debugging. Some possibilities for replacing this behavior might involve repurposing sensor motes again to a different end. Motes could store instrumentation output in memory, and send these data packets through the network to a base station for collection. The emulator also has the ability to control the evaluation and execution speed of the simulation itself, which cannot be done on a real sensor mote. Facilitating this functionality is a technically involved endeavor which is separate and divergent from the development of Policy IDE and requires a rethinking of the core process model of tinyOS.

Developing Policy IDE for the generic TOSSIM emulator in comparison with specific sensor mote platforms is another potential avenue of research. Future work concerning exploration of instrumenting sensor motes directly and conducting real-time debugging would need to be limited to a specific series of devices. The tool-chain provided for compilation of nesC code on each specific platform is limited to predefined parameters packaged with the tinyOS software development kit. These parameters allow for code to be built on specific platforms only, cross-compiled for each device. While it is likely that most instrumentation code added for usage with TOSSIM would not affect the stability or compilation of nesC code and policy development, this has not yet been tested. Indeed, without the prerequisite software modules for interacting with the sensor motes, the feasibility of interactive debugging and policy development on platforms outside of the emulator is a difficult problem to approach. Exploring this further would require that these prerequisites are addressed first.

Conversely, there are several points for improvement on new iterations of Policy IDE. One improvement would add timestamps to log messages. This could more accurately display chronological information for the execution trace. The IDE could show debug log information alongside the simulation controls to enable developers’ access to the debugging information without switching interface tabs. Another valuable improvement, such as a ‘step forward’ button, would allow developers to control the simulation with finer time-stepping controls. Moreover, other control fields can be added to the IDE, which would allow control over the simulation environment within the tool itself. These parameters may include the WSN topology, or internode connection parameters such as signal noise. Further improvements could add the ability to load policies in batches, in addition to individual policies. Policy storage and loading could use a text intermediary format, which would aid in creating batch policy files or policy profiles. The ability to view which policies that exist in memory would also be useful.
References