Proceedings Paper:
Tunc, H, Taddese, A, Volgyesi, P et al. (2016) Web-based Integrated Development Environment for Event-Driven Applications. In: SoutheastCon 2016. SoutheastCon 2016, 30 Mar - 03 Apr 2016, Norfolk, UK. IEEE. ISBN: 978-1-5090-2246-5. ISSN: 1558-058X. EISSN: 1558-058X.

https://doi.org/10.1109/SECON.2016.7506646

Web-based Integrated Development Environment for Event-Driven Applications

Hakan Tunc, Addisu Taddese, Peter Volgyesi, Janos Sallai, Pietro Valdastri, Akos Ledeczi

The Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee 37212
{hakan.tunc, peter.volgyesi, janos.sallai, akos.ledeczi}@vanderbilt.edu

STORM Lab
Department of Mechanical Engineering
Vanderbilt University
Nashville, Tennessee 37212
{addisu.z.taddese, pietro.valdastri}@vanderbilt.edu

***Abstract*—Event-driven programming is a popular methodology for the development of resource-constrained embedded systems. While it is a natural abstraction for applications that interface with the physical world, the disadvantage is that the control flow of a program is hidden in the maze of event handlers and call-back functions. TinyOS is a representative event-driven operating system, designed for wireless sensor networks, featuring a component-based architecture that promotes code reuse. In this paper, we present a web-based model-driven graphical design environment for TinyOS that visualizes the component hierarchy of an application, and captures its event-based scheduling mechanism. In contrast with existing visual environments, our representation explicitly captures the control flow of the application through events and commands, which makes it easier to understand the program logic than studying the source code. The design environment supports two-way code generation: mapping the visual representation to TinyOS source code, as well as building visual models from existing sources.**

I. Introduction

The key building blocks and the enablers of any IoT (Internet of Things) and most CPS (Cyper-Physical Systems) applications are compact, low-power, inexpensive, yet responsive sensor and actuator devices with built-in intelligence. Such requirements and the need for building highly efficient embedded solutions are in contrast to the developers' desire to use structured, composable and scalable software architectures. In fact, two fundamental computational models are used extensively: software threads with blocking services [1] and time-triggered threads. Software threads aim at the composability problem in the memory space (i.e. memory requirements and separation of state variables), while the time-triggered [2] approach has nice guarantees in the time domain. Unfortunately, one cannot mix and match these methods at the same time, and both sacrifice efficiency in their *integration domain*. Threads require pre-allocated exclusive memory stacks, which puts the burden on the system designer to estimate the maximum needs of each task at design time. Context switching is another—yet, relatively small—cost of threads, which affects interrupt latency and overall responsiveness. Time-triggered systems employ static periodic scheduling of tasks, thus estimating the worst-case execution time (WCET) is essential for a successful design. Due to static scheduling, and the lack of interrupt-driven behavior, the responsiveness of these systems is bounded but suboptimal. Conservative over estimation of the required resources in both models leads to significant waste.

Strictly event-driven software architectures on the other hand sacrifice system integration and software composition assurances while aiming at optimal memory, time and power use. Unfortunately, the *event-driven* terminology is used liberally for many vaguely similar computational models. In this paper we use the term for describing software where relatively short tasks always run to completion, with no preemption and context switching among tasks. Such tasks can be scheduled for execution by other tasks or by interrupt service routines. Interrupts are enabled during task execution and can optionally be nested. In case there are no pending interrupts or runnable tasks, the system enters a sleep mode, from which only new interrupts can wake it up again. Such system can be implemented by a *big loop* of task function calls or using a static array or dynamic queue of task/function pointers.

It is easy to see how this architecture strives to run only the necessary piece of code as fast as possible in response to external—including previously scheduled timer—events. The price of such architectural choice is mostly paid by the system designer and developer: due to the *run-to-completion* and *non-blocking* requirements the state space of the application has to be explicitly maintained and because there are no constraints on function calls from interrupt and tasks (deep call stacks), the system architect should have a relatively clear picture of the overall behavior of the entire system.

It is interesting to notice that very similar event-driven approaches have gained popularity on the other end of the

computational spectrum: high-end web servers almost completely abandoned the *thread-per-client* design in favor of event-based processing. In fact, the JavaScript run-time model, which is at the heart of all modern web applications builds on asynchronous event-driven scheduling, exclusively. The previously described burden on the JavaScript developer is also well-known and somewhat mitigated by function closures—a highly dynamic language feature which, unfortunately, cannot be easily adapted to resource constrained platforms.

Thus, the challenge in supporting event-driven software development is to provide tools and frameworks with zero run-time overhead or performance penalty. TinyOS improves structural composition, provides a clear task model and keeps track of interrupt initiated vs. task-only function contexts to detect potential race conditions on variable access [3]. However, it provides little support in the behavioral aspect of the software components. In this paper we propose a lightweight visual approach to mitigate this problem and augment TinyOS by keeping track of the internal control flow within the software components. The model-based environment is supported by code generators and parsers for establishing a live connection between the visual representation and the implementation-level source code.

The next section reviews the existing similar development environments and contrasts our tool with them. Section III introduces TinyOS, as well as WebGME, the web-based modeling platform that is the foundation of our model-based TinyOS development environment. Section IV outlines our approach for depicting event-based programming and TinyOS model representation. Section V describes the tool we developed in detail. Section VI develops an example app using the tool. Section VII reveals planned future work and the shortcomings of our approach with a conclusion.

II. Related Work

Although a number of TinyOS development environments are available, most of them are discontinued or not compatible with recent versions of the operating system.

YETI [4] is an eclipse plugin for TinyOS 2.1. It supports syntax highlighting, code completion, error detection, refactoring, debugging, and hyperlink navigation across files and to definitions. Although YETI generates a component graph of the applications, it does not support visual editing. The plug-in was last updated four years ago.

Viptos (Visual Ptolemy and TinyOS) [5] is a graphical development and simulation environment for TinyOS based Wireless Sensor Network applications. Viptos integrates with TOSSIM, a TinyOS simulator. Viptos allows developers to visually build configuration components by dragging and dropping, as well as by creating connections between them and transforming them into nesC programs that can be compiled and downloaded. It can parse existing TinyOS components and represent them visually. Although Viptos can be used for configuration development, it is mostly focused on simulation of TinyOS programs. Viptos supports the earlier version of TinyOS 1.x and the development of Viptos seems to be discontinued since 2006.

GRATIS II [6] is a graphical development environment for TinyOS. It is built on the Generic Modeling Environment (GME), the predecessor of WebGME, and uses models and ports to represent components, interfaces and events. The operating system components are available for reuse. GRATIS II generates TinyOS source code from a visual representation of configuration components. It applies constraints specified in the Object Constraint Language (OCL) to keep the applications valid. Its latest release supports only TinyOS 1.x.

Tei, *et al.* [7] have developed a tool to ease the development process of WSNs. In this tool, a set of predefined components are used to create applications and TinyOS source code is generated from template files. However, this tool currently supports only simple monitoring applications. Moreover, three different modeling languages are used to create applications, which presents a steep learning curve for new users.

Our tool differentiates itself from other related tools. For example, rather than adding multiple abstraction layers, we focus on creating the a single intuitive abstraction layer on top of nesC and TinyOS to make it easier to conceptualize and develop applications.

Since our tool resides on the web, users do not need to install any applications or set up a toolchain. Web browsers are more accessible to end users. Unlike desktop programs, web applications can be used from a wide range of operating systems and devices. Updates to web applications are quickly propagated to users as opposed to desktop programs that require the user to update them regularly. New web technologies are creating new workflows and ways of collaboration that are increasing the productivity of developers. Our tool provides the benefits of new technologies to all users including those new to TinyOS. We believe that the TinyOS developer community can benefit from an online collaborative tool that removes some of the barriers of entry for novice users.

Another important differentiation of our tool is the revision control system. WebGME's revision control capabilities are mentioned in Section III. WebGME also has a constraint mechanism which makes it possible to enforce domain, as well as TinyOS related constraints. In addition, it can be used to enforce design patterns and conventions envisioned by those who developed the corresponding components.

III. TinyOS & WebGME

A. TinyOS

TinyOS is an event-driven operating system for small devices. Its primary design objectives were support for complex, concurrent programs, e.g. sensing and forwarding messages at the same time, as well as promoting code reuse through defining a component model.

At its core, TinyOS is a classical event-driven operating system. Once booted, the scheduler runs the event loop that dispatches *tasks* from the event queue. Tasks are non-periodic, and always run to completion, i.e. there is no preemption other than interrupts. Tasks are *posted* (i.e the deferred execution of

the task is requested) by interrupt service routines or other tasks. A task is implemented as a function, and posting a task is essentially equivalent to adding the tasks function pointer to the event queue.

What makes TinyOS stand out from the crowd is the way application code is structured. The source code of tasks, functions, callbacks, and static variables that are logically related are organized into a *module*. Modules interact through bidirectional *interfaces*, that include *commands*, which are function calls, such as SendMsg.send(), as well as *events*, which are callbacks, such as SendMessage.sendDone(). Modules may provide and/or use a given interface. When a module *uses* an interface, it can call the interface's commands and must implement the *event* handlers (callbacks) defined in the interface. Conversely, when a module *provides* an interface, it must implement the corresponding commands, and may *signal* (call) the events the interface defines.

To promote code reuse, as well as programming abstractions, TinyOS allows for composing modules into *configurations*. Configurations are comprised of multiple components (modules or configurations), that interact through their interfaces. Configurations specify *wirings* that connect the provided interface of one subcomponent to the used interface of another. Alternatively, an interface of a subcomponent may be exposed by the enclosing component, as well.

In TinyOS, operations that one may request through an interface, e.g sending a radio message, are often split phase. The completion event of an operation is signaled in a new task or interrupt context. This has two important implications. First, since the stack is unrolled by the time the event is called back, local variables are not retained between command and event invocations. Second, the logical control flow within a module is not linear, but rather a series of command, event, and task invocations.

TinyOS applications are programmed in the nesC language, a superset of C that supports modules, configurations, wiring, interfaces, commands, events, tasks as first class language elements. A TinyOS application is always specified as a top-level configuration that is a hierarchical composition of modules and configurations, most of which come from a library of components that is shipped with the OS, while the rest are custom modules and configurations that implement the application-specific logic and glue code.

The nesC compiler is a source-to-source compiler that, in essence, traverses the component hierarchy, resolves the wirings, and emits a monolithic platform-specific C source file which will be compiled to a binary image by a C compiler. As a side product of the compilation process, nesC can optionally output an XML file with detailed information on the component structure of the TinyOS application, as well as custom nesC annotations, variables, event and command invocations.

B. WebGME

Our tool is based on WebGME [8], an online collaborative environment for designing complex computational systems. WebGME benefits from the features that web and cloud infrastructures, modern web browsers and HTML5 offer. In addition, it provides most of the necessary infrastructure on which we have built our design environment.

WebGME is a visual design environment for model driven development [9]. It uses meta-models, which specify a visual domain-specific language and corresponding model. The model represents the structure and behaviour of the systems being designed. Every object and connection, which in itself is an object, is represented by a graphical object in the WebGME canvas. WebGME provides different perspectives to manipulate models and meta-models. In addition, WebGME allows custom domain-specific visualizations and plugins which we have utilized for our custom TinyOS design environment.

WebGME has a built-in revision control mechanism. Creation of new components, changes of attributes and even position changes of objects are versioned. Since all changes are stored in a cloud database, users can revert their project to an earlier version or explore its development history. WebGME also provides an easy-to-use branching mechanism, which, combined with the overall revision control system, gives users an opportunity to experiment without the risk of losing their progress. It is especially useful for novice users since they know they can revert their changes any time or they can create an experimentation branch.

IV. Representation

Among the other integrated development tools for TinyOS, there is very little tool support for event-based architecture. On the other hand, we model this aspect of the operating system in an informative visual representation so developers have an alternative view to the source code for their application's implementation logic. In addition, we show the local variables used in a *module* as well as their read-write accesses from the *interfaces*. This gives another view to evaluate the implementation of a system.

The component based nature and hierarchical structure of TinyOS fits harmoniously with WebGME's modeling framework. First, we devised a visual representation of the TinyOS language (nesC) using the WebGME meta-modeling language. We then added extra modeling concepts to emphasize the event-based aspects of TinyOS and to show variable access patterns. In the following sections, we present more details about the formalism.

A. How to represent event based programming

We visualize the program flow in TinyOS using states and transitions between states triggered by *events* or *commands*. The states are represented as components and they are connected together through ports representing either *events* or *commands*. Connections indicate the possible ways to transition to another state.

In TinyOS, the logic of the applications is implemented in *module* components. The *events* of used interfaces and the *commands* of provided interfaces of a *module* are implemented as functions written in nesC. These functions can be

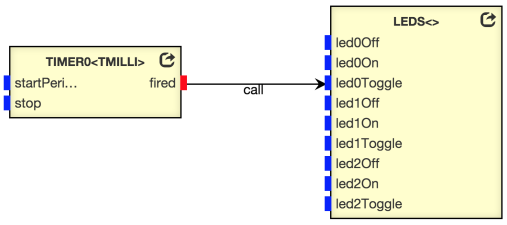

Fig. 1. Simple Event Triggered Call

invoked with the keywords *signal* and *call*, respectively. The component and interface based structure of TinyOS has great advantages in representing the flow of an application visually. Since all of the functionality and flow of an application is stated with interfaces, we capture the 'uses and provides' interfaces as states. The *command* and *event* functions of the interfaces are represented as the ports of the interfaces.

In Fig. 1, we show a simple state flow between two interfaces. Yellow boxes, blue ports, and red ports represent interfaces, commands, and events respectively. When Timer0's timer fires, TinyOS dispatches the *fired* event. This event calls the led0Toggle function of the Leds interface. This visual representation makes TinyOS programming logic easier to understand and modify. The visual representation is an easy way to conceptualize the call graph of a module compared to examining the source code and following event dispatches and function calls.

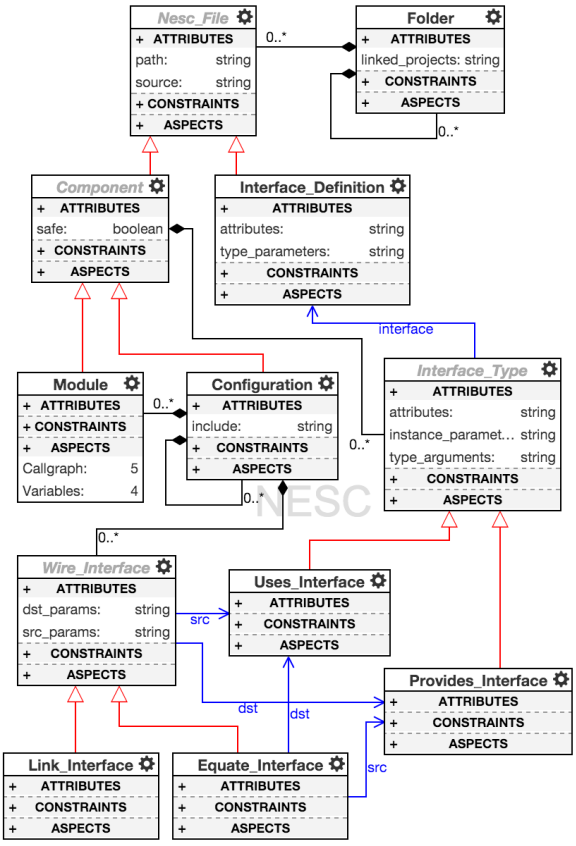

Fig. 2. Meta Language for nesC

B. Meta Model

We capture the nesC grammar specifications with WebGME's meta modeling capabilities. Our goal is to come up with the most natural set of visual elements while capturing every necessary detail of the language specification. Most of the elements are represented as objects while some of them are included as attribute fields. In some cases, more than one object represents a language component in order to provide a better understanding of the application in a visual view. For example, we use two different base types for *uses interface* and *provides interface* instead of using an attribute field to choose the type. In this way, it is easier to create the object and make it visually different.

In Fig. 2, we show the meta language of fundamental entities of nesC in WebGME's meta language representation. WebGME uses Unified Modeling Language (UML) class diagrams for language definitions. The red lines with triangle arrows represent inheritance, black lines with diamonds represent the containment relationships, blue lines with arrows represent pointers (associations). By convention, if an object has both a source (src) and a destination (dst) pointer, it is visualized as a connection in the modeling language. Objects with gray titles are abstract types. We cropped some parts of the language in this figure to save space, but the complete meta language can be studied from the project web page. This figure should be used as a reference to understand the meta level relations between objects for the remainder of this section.

Each *configuration, module*, and *interface-definition* of TinyOS, in addition to *uses* and *provides interfaces*, is represented by a WebGME object in a visual canvas. Within a configuration or module, there are *uses* and *provides* interfaces, *configuration* and *module* components, and *link* and *equate* connections (wires). These interfaces, components, and wires are designated by color. *Uses* and *provides interfaces* are yellow and green. *Configuration* and *module* components are light blue and orange. *Link* and *equate* wirings are continuous blue and dashed red lines, respectively.

Interfaces that belong to a particular configuration or module are shown as ports on a WebGME object along with its name. Components' interfaces are visible as ports. Connections between the interfaces and ports indicate wiring. WebGME supports prototypical inheritance for models. Here model instances are utilized to refer to components instantiated elsewhere in the object hierarchy. Clicking on instances takes the user to the original interface or component definition.

As an example, visual representation of the *MainC* configuration can be seen in Fig. 5. While SoftwareInit and Boot interfaces are exported for applications, TinySchedulerC and PlatformC are wired to provide implementation of the interfaces Scheduler and PlatformInit.

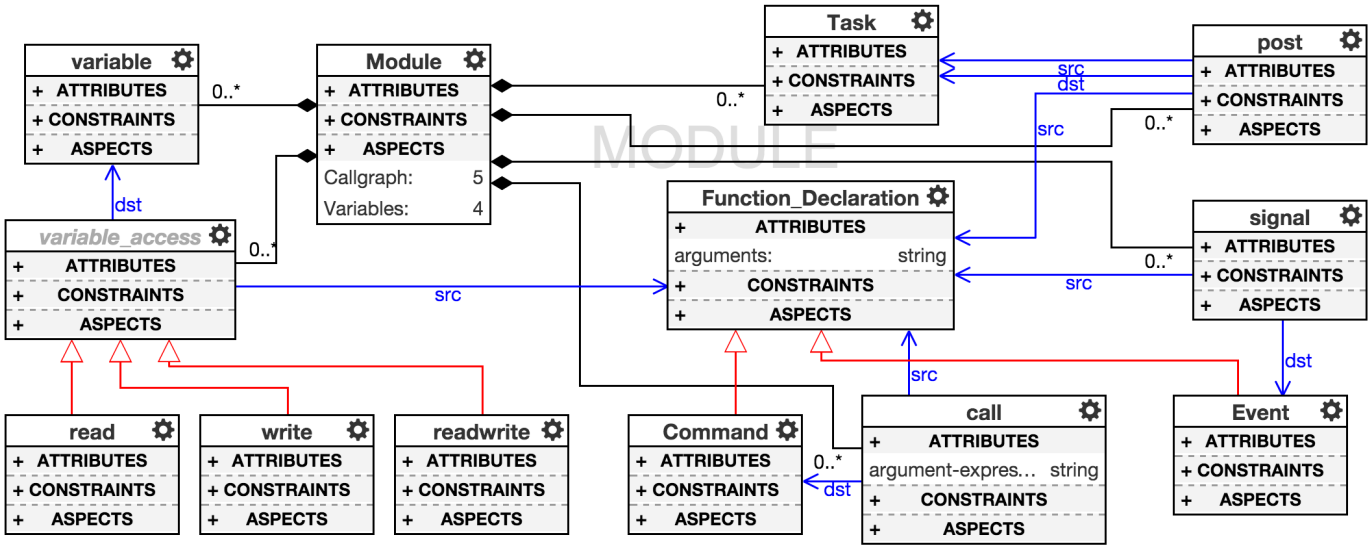

Fig. 3. Meta Language for nesC Modules

Uses interface and *provides interface* inherit from *interface type* object. *Interface type* is defined as an abstract type. Thus, it is not possible to create an instance of it while developing a model. We defined a pointer from *interface type* to *interface definition* named *interface* in order to be able to reference the original definition. In Fig. 5, there are *uses interface* 'SoftwareInit' and *provides interface* 'Boot'. The *interface* reference is represented as a small icon on the top right of these two objects. When a user double clicks this icon, the user is taken to the definition of the used or provided interface.

The internal structure of a *module* is the most versatile among the models we have used. The meta language definition of modules can be seen in Fig. 3. Modules include *interfaces*, *tasks*, *variables*, a call graph (call, signal, post), and variable access patterns.

Uses and *provides interface* objects reference an interface-definition object in order to have access to the function declarations of the interface. Within *module* implementations, *uses* and *provides interfaces* show the function declarations (*events* and *commands*) as ports as can be seen in Fig. 1.

If a *module* component has any *tasks* defined, they are also represented as WebGME objects which are displayed using a 'gear' icon inside the module visualizations. To capture the call graph of a module, different types of connections can be created between function objects. Function objects are either ports of *interfaces* or *tasks*. There are three types of function calls: *call* for *commands*, *signal* for *events*, and *post* for *tasks*. They are all visualized as black lines with an arrow and they represent the call graph of a module. The type of connection is labeled with these names. An example can be seen in Fig. 4.

Local variables used in a module are represented by a circle. These variables are connected to the *tasks* or interface functions (*commands or events*) with connections: read, write or readwrite. The connection type is based on whether or not the variables are accessed, assigned, or both. We modified the TinyOS compiler to retrieve call graphs and variable accesses of modules.

When a module has a slightly complex internal structure with a number of variables, the representation of the module has many connections which may lead to a complicated view. We have used WebGME's *aspect* feature to define two additional views for a modules' internal structure: call graph and variables. Aspects allow us to define which subsets of contained objects to show together. Within a module, it is very easy to toggle between the three aspects defined: all (which is the default view and shows everything), call graph and variables.

Under the hood, we heavily use the inheritance functionality of WebGME while defining the meta-model of the modeling language. For instance, *configuration* and *module* directly inherit from *component* which has the field *safe*, while both *component* and *interface definition* inherit from *nesc file* which has *path* and *source* fields. By relying on inheritance, the meta-model development becomes more maintainable without repetition and it is less error prone.

When an object is selected in WebGME, its pointers can be accessed from the property editor. The property editor is located on the bottom right of the window as can be seen in Fig. 4. In addition to TinyOS components, we defined a *Folder* object in WebGME to be able to keep the original folder structure of the TinyOS source code (TOS). As a result, the

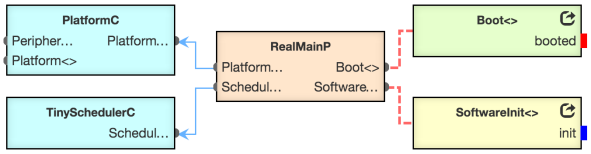

Fig. 5. Visual Representation of MainC

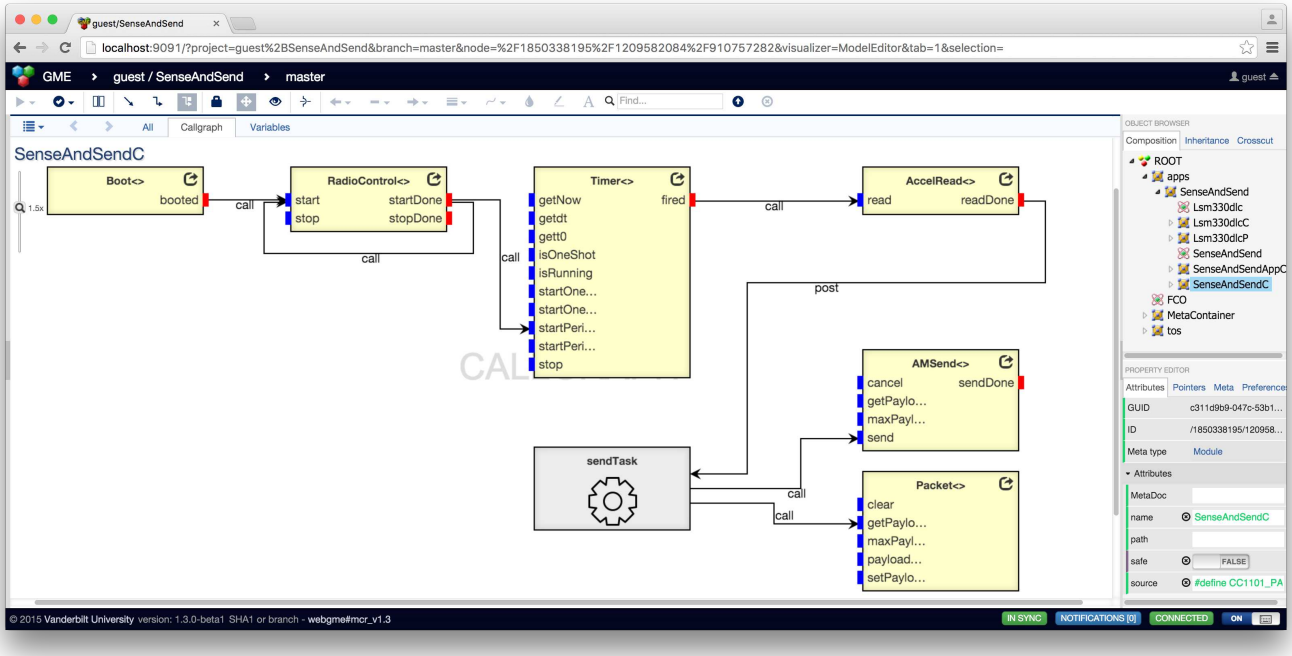

Fig. 4. Visual Representation of a Module: SenseAndSendC

objects can be browsed in the TOS structure and experienced developers can easily find the desired components.

The visual editor allows an intuitive representation of the components and the connections between them. It increases readability and understandability of applications with easier conceptualization. In addition, the WebGME constraint checking mechanism prevents wiring between incompatible interfaces which eliminates some of the possible compile time errors. It is not only helpful for novice users but also for experienced users because it allows them to strictly focus on their design.

V. Software Design Environment

We extended WebGME with plugins, software components that provide domain-specific features, and a custom decorator to enhance the visualization of TinyOS applications. The plugins are triggered from the user interface and are easy to use. The development and build workflow are shown in Fig. 6. All computation that need to use the TinyOS toolchain is handled on the server side and the resulting binaries are sent to the client. The XML output of the nesC compiler, mentioned earlier, has been used to access the information of an application's structure whenever necessary. The following sections cover the different aspects of the design environment [10].

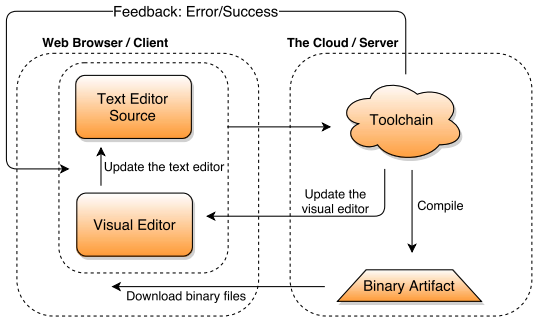

Fig. 6. Development and Build Workflow

A. Decorators

We extended the WebGME built-in decorators to serve our needs for visualization and functionality. Custom additions to the tool allow users to run plugins, open the editor, edit the source code, compile and download the app as seen in Fig. 7.

B. Integration of Visual and Text Editor

1) Code Generation: Code generation is an integral part of the tool. Although the configurations and applications can be designed with the visual interface, we need to feed the compiler with an error-free nesC code to compile the application. In addition, the code generation plugin allows a user to generate nesC code of a component to study its source code. The code generation plugin parses the WebGME objects and uses a template engine to generate its source code. The generated source code is stored as an attribute of the corresponding component.

2) Model Generation: Users who prefer working on the source code can use the text editor view of the tool as can be seen in Fig. 7. We want to make it possible for users to use both graphical and classical ways of developing an application.

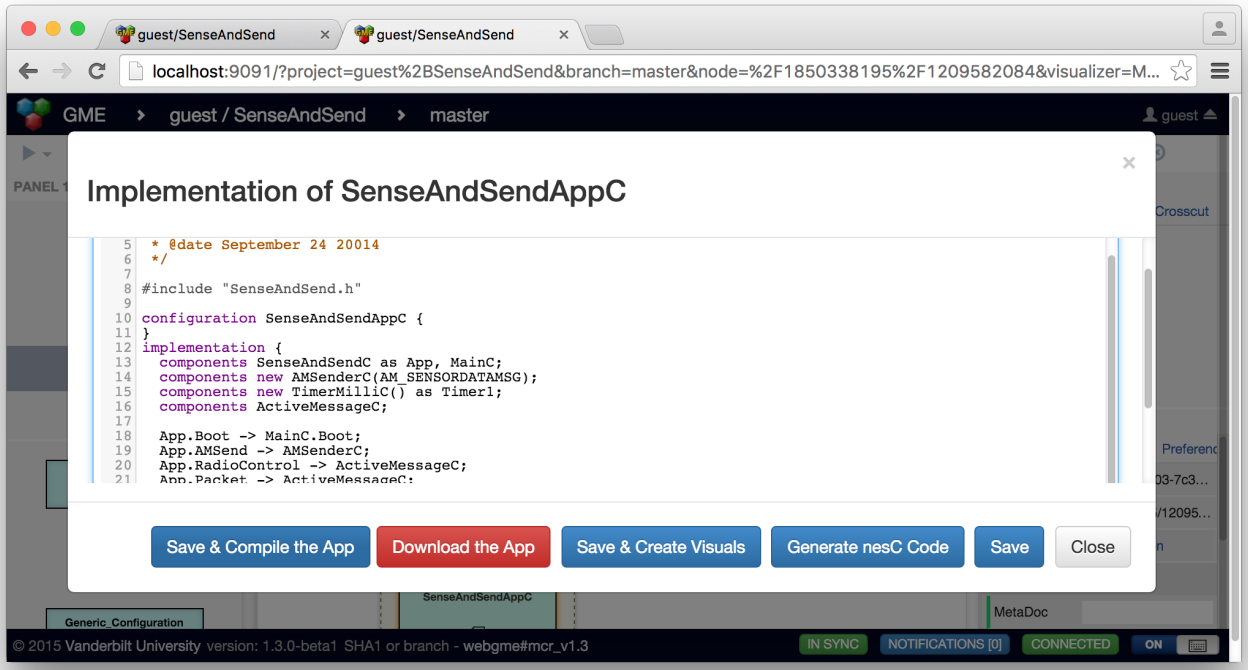

Fig. 7. Source Editor

Each user, expert or novice, can switch between text and visual modes as they are developing their application and continue their work by updating the corresponding representation of the component. Users can update the visual or textual editor by using the buttons at the bottom of the editor.

For module components, model generation is a one-way transformation from the source code to WebGME model. The visual representation of modules is read-only and helps users understand the program logic of the component.

The model generation plugin creates the module, (local) variables and their access patterns from the interfaces within the module representation. The access pattern is shown whether it is read, write or read-write. This visual representation makes it easy to detect undesired variable modification. Since a TinyOS program's states are captured with its components and their local variables, local variable analysis will give us the chance to get statistics about how resources are consumed from individual components. In that way, the developer can update their code to use the resources more efficiently; this is particularly important for embedded systems due to limited amount of memory.

C. Build System - Compilation

Our tool allows users to compile and download the app from their web browsers without installing any toolchains. The tool compiles the applications on the cloud and sends back a zipped collection of artifacts which can be downloaded by the user and installed on the target device. Once the user compiles the app from the editor, *Download the App* button appears. If the user recompiles the app after any changes, the tool updates the version number of the downloadable. The installation of the downloaded files onto the target device requires device specific tools to be available on the users computer. This is a limitation of our tool that we are currently working on to address by way of providing a hardware platform compatible with available technologies that allow programming of devices from web browsers.

1) TinyOS Populator: In order to create the WebGME representation of TinyOS library, we wrote a script which utilizes the XML output of nescc (compiler for nesC). Since there are many platforms and hundreds of TinyOS elements (components and interfaces) for each platform, the WebGME representations of the objects are created for a specific platform by the script. As a result, each TinyOS component and interface is represented by a corresponding WebGME object in the original folder structure of the source code. This library of TinyOS components is available in the object browser and does not need to be generated by the user.

2) App Importer: This plugin is used to import existing TinyOS applications into our tool. It works in a similar fashion to the TinyOS Populator in terms of creating WebGME objects. This plugin can be used with an empty project as it creates the used TinyOS objects during the import process.

VI. Example: Sense and Send Application

To create an app, a user must create a configuration object in a user space, e.g., *apps* folder under the *ROOT* object. TinyOS source code is accessible with every project as a library. Whenever a user needs to use a TinyOS component or interface, they need to locate the component in the object browser and create an instance of it in the user's configuration object. WebGME will automatically create the full-fledged component with its interfaces. Wiring is indicated by connecting ports of components or interfaces with a mouse click.

As an example, we are using a sense and send application we have developed to test our hardware and software environments. It is a simple yet typical medical capsule robot application that senses its environment and sends the collected data through a radio. We used the AppImporter plugin to import the helper components and convert the project to the WebGME environment. To create an app, we create a configuration object and name it SenseAndSendAppC. We create a module of SenseAndSendC within the app object. Since the SenseAndSendC component is using a number of interfaces, we create instances of components that provide these interfaces. Then, we connect the interfaces through the ports of components with link wires. At this point, we can compile and download the application by opening the SenseAndSendAppC editor.

Visual representation of the SenseAndSendC module call graph can be seen in Fig. 4. Although this representation is view only, it provides an easier to understand overview of the application with the call graph. In Fig. 4, we only show the call graph aspect of the model to highlight the business logic.

VII. Conclusion & Future Work

Currently, the visual representation of autogenerated modules does not adequately reflect the business logic of the corresponding application. It will be useful for the users if the tool creates a smarter layout which do not necessarily require the users to move objects to understand the flow of the application. Furthermore, using an existing component requires finding its location in TinyOS using the object browser. We would like to develop a functionality that searches the existing components and gives the chance to create them right from the visual editor. It would be very useful for users to be able to list and search interfaces, for instance, without looking through the object browser.

Future work also includes improving the static error detection and prevention mechanism of the visual editor. With an improved approach, users will get immediate feedback on their application reducing development time. One use case of this is the prevention of incorrect wirings in configurations.

In TinyOS, once an application is developed, the application can be compiled easily to other platforms, as long as the components are supported by the hardware. However, in our tool, the user directly uses the specific platform's component implementations while they are developing their application. Therefore, porting the application to different platforms is not as easy as it is in the classical TinyOS development process. It will be possible to address this by automatically repopulating the TinyOS library and the application components with a plugin for the desired platform.

The design environment needs to be further improved by giving the user messages from the TinyOS toolchain. This will help the user with debugging his application. It might be more beneficial if the user has access the warnings and errors as he develops his application.

We have described the design and implementation of a web based development environment for TinyOS applications. We believe this design environment reduces the barrier of entry for novice users while offering benefits to advanced users in terms of convenience and higher level abstractions not available directly in TinyOS. While the advantages of this design environment are clear from the authors' experience, more work is needed to quantify its utility for users. Metrics for assessing the success of this tool on novices and advanced users need to be defined and methods for obtaining such metrics from users need to be devised. Additionally, models that provide even higher levels of abstraction in various domains need to be supported to seed the growth of the repository of models that will later be supported by the community.

VIII. Acknowledgment

This material is based upon work supported by the National Science Foundation under grants number CNS-1239355 and IIS-1453129, as well as an NSF Graduate Fellowship. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] E. Lee, "What's ahead for embedded software?," *Computer*, vol. 33, pp. 18–26, Sep 2000.

[2] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, pp. 112–126, Jan 2003.

[3] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, *et al.*, "Tinyos: An operating system for sensor networks," in *Ambient intelligence*, pp. 115–148, Springer, 2005.

[4] N. Burri, R. Flury, S. Nellen, B. Sigg, P. Sommer, and R. Wattenhofer, "Yeti: an eclipse plug-in for tinyos 2.1," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pp. 295–296, ACM, 2009.

[5] E. Cheong, E. A. Lee, and Y. Zhao, "Viptos: a graphical development and simulation environment for tinyos-based wireless sensor networks," in *SenSys*, vol. 5, pp. 302–302, 2005.

[6] P. Völgyesi, M. Maróti, S. Dóra, E. Osses, and A. Lédeczi, "Software composition and verification for sensor networks," *Science of Computer Programming*, vol. 56, no. 1, pp. 191–210, 2005.

[7] K. Tei, R. Shimizu, Y. Fukazawa, and S. Honiden, "Model-driven-development-based stepwise software development process for wireless sensor networks," *Systems, Man, and Cybernetics: Systems, IEEE Transactions on*, vol. 45, no. 4, pp. 675–687, 2015.

[8] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Levendoszky, and Á. Lédeczi, "Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure," *Proceedings of MPM*, p. 41, 2014.

[9] A. Lédeczi, A. Bakay, M. Maroti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments," *Computer*, vol. 34, no. 11, pp. 44–51, 2001.

[10] Pillforge, "Tinyos development environment." https://github.com/pillforge/mcr_ide, 2015.