Component-Based Development of Networked Embedded Applications

Péter Völgyesi and Ákos Lédeczi
Institute for Software Integrated Systems, Vanderbilt University
{peter.volgyesi, akos.ledeczi}@vanderbilt.edu

Abstract

Networked Embedded Systems (NEST) are large-scale distributed systems with resource limited processing nodes tightly coupled to physical processes via sensors and actuators. These strict resource constraints mandate thin application-specific operating system and middleware layers. Component-based development is an enabling technology in this arena. We present a model-based approach to the development of applications based on TinyOS, an important NEST platform. OS and application component interfaces along with their interdependencies are captured in a graphical environment and the glue code that ties together the application and OS components are automatically generated. A fully functional sophisticated modeling and code generation environment was developed in one man-month. This is due to the model integrated technology applied in the tool development.

1. Introduction

Networked Embedded Systems (NEST) are large-scale distributed systems with resource limited processing nodes tightly coupled to physical processes via sensors and actuators. NEST systems are distributed, but the nodes must achieve a centralized goal cooperatively. Typical NEST applications include large-scale active noise and vibration control, cooperative sensor networks, micro-satellite constellations and smart structures.

NEST nodes typically have limited computing power and small amounts of memory. They must consume as little power as possible. Communication is noisy and its bandwidth is limited. The individual nodes and communication channels are inherently unreliable, yet the overall system needs to be robust. These requirements mandate novel systems and software engineering techniques.

A typical NEST application running on each node can be separated into the three layers depicted in Figure 1. The local operating system manages the underlying hardware infrastructure including interfacing with the physical world through sensors and actuators. The distributed nature of computation implies that each node should be equipped with sophisticated middleware —a kind of distributed operating system that provides global services for the applications, such as message routing, broadcast, consensus, group membership, etc. The actual application code utilizes both the OS and the middleware layers.

What distinguishes a NEST application from a traditional one is that neither the operating systems nor the middleware can be the usual, heavyweight, monolithic component that contains all services for all applications. The strict resource constraints, especially the memory limitations, mandate application-specific, thin operating system and middleware layers.

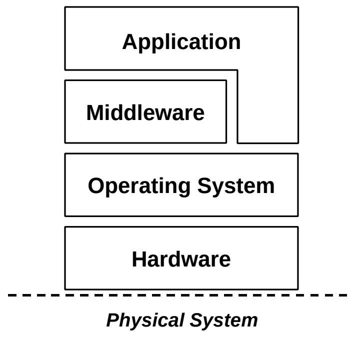

Figure 1. Layered Architecture

A component-based approach to the problem is quite natural. Specifically, the computational model of TinyOS, a highly modular, customizable operating system for tiny embedded devices developed at UC Berkeley (http://tinyos.millennium.berkeley.edu), is an ideal fit for the task at hand. However, the specification of component interfaces, hierarchical component dependencies and interactions using the current text-based environment is cumbersome and error-prone. This paper introduces GRATIS, a graphical development environment for TinyOS that provides an intuitive visual interface and automatic code generation capability for the development of TinyOS-based applications.

The rest of the paper is organized as follows. The next section describes the software infrastructure used to develop GRATIS. Next, a typical hardware platform for NEST in general and TinyOS in particular is presented. This is followed by a brief introduction to TinyOS. Finally, GRATIS is described in detail along with a simple case study.

2. Model Integrated Computing

The Graphical Development Environment for TinyOS (GRATIS), which is in the focus of this paper, was developed using Model Integrated Computing (MIC) technology [1]. A brief summary of MIC follows.

MIC employs domain-specific models to represent the system being designed. These models are then used to automatically synthesize the applications and/or to generate inputs to analysis and/or simulation tools. This approach speeds up the design cycle, facilitates the evolution of the application, and helps system maintenance, dramatically reducing costs during the entire lifecycle of the system.

MIC is implemented by the Generic Modeling Environment (GME), a metaprogrammable toolkit for creating domain-specific modeling environments [2]. GME employs metamodels that specify the modeling paradigm of the application domain. The modeling paradigm contains all the syntactic, semantic, and presentation information regarding the domain – which concepts will be used to construct models, what relationships may exist among those concepts, how the concepts may be organized and viewed by the modeler, and rules governing the construction of models. The modeling paradigm defines the family of models that can be created using the resultant modeling environment. The metamodels specifying the modeling paradigm are used to automatically configure GME for the domain. An interesting aspect of this approach is that GME itself is used to build the metamodels themselves using a formalism based on the UML class diagram notation [3].

GME is used primarily for model-building. The models take the form of graphical, multi-aspect, attributed entity-relationship diagrams. The static semantics of a model are specified by OCL constraints [4] that are part of the metamodels. They are enforced by a built-in constraint manager during model building time. The dynamic semantics are applied by the model interpreters, i.e. by the process of translating the models to source code, configuration files, database schema or any other artifact the given application domain calls for.

This approach fits component-based software development very nicely. The interface of the individual components can be modeled along with a link to their implementation. The model editor can enforce the composition rules to make sure that only valid component assemblies are allowed. Finally, model interpreters can generate the glue code that ties the final system together. These concepts are perfectly demonstrated by GRATIS described later in the paper.

3. Target Hardware Platform

One of the NEST experimental target platforms is a network of tiny devices, called the motes. Each mote is about 2 inches by 4 inches. It is powered by a pair of AA batteries. The heart of the device is the ATmega 103L 8-bit micro-controller. It has 128 Kbytes of flash (instruction) memory and 4 Kbytes of SRAM (data memory) on-chip. The micro-controller has 8 A/D channels and an UART built-in.

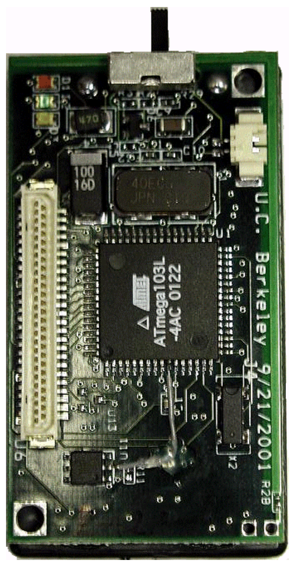

Figure 2. A Berkeley mote

A coprocessor provides access to the on-board 4-Mbit EEPROM that can be used for data-logging or even for reprogramming the CPU on-the-fly. This can be done via the radio unit that runs at 900 MHz and has a maximum transfer rate of 50 Kbits/sec. The radio has a programmable gain. It’s maximum range is about 30 feet. The unit also sports 3 LEDs for debugging and demonstration purposes. A simple programming board can be used to download programs via a standard PC parallel port and/or to communicate with a mote via the serial port.

4. The TinyOS Operating System

The unique characteristics of these motes lead to the development of TinyOS, a component-based, highly configurable embedded operating system with a very small footprint [5]. A TinyOS instance consists of a set of interconnected components scheduled by a tiny, FIFO-based scheduler. Components communicate with each other through commands and events. Commands propagate downward; they are issued by higher level components to lower level ones. Events propagate upward; they are signaled by lower level components and handled by higher level ones as shown in Figure 3. The lowest level of events are generated by the hardware itself in the form of interrupts.

A component consists of a set of tasks, event- and command handlers and a frame, a statically allocated piece of memory that stores the state of the component [5]. TinyOS does not support dynamic memory allocation, a significant restriction, but one that buys efficiency and simplicity. The hardware does not provide much memory to allocate in any case.

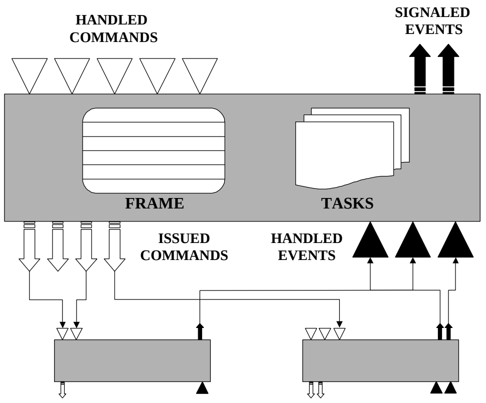

Figure 3. TinyOS concepts

Commands are typically handled by modifying the state of the component, possibly posting a task for later execution and possibly issuing commands to lower level components. An event handler can also modify the state of the component, signal higher level events or call lower level commands. Note that commands cannot signal events to avoid cycles. Tasks are the worker bees of TinyOS. They can issue commands, signal events and post other tasks. Tasks are intended to do a short amount of processing and return. They can only be preempted by events. This task model enables TinyOS to have a single stack.

Possibly the best feature of TinyOS is its component-based configurability enabling the creation of highly application-specific operating systems. In addition to its C implementation, each component has a special file (.comp) that declares what commands and events it handles and also what commands it issues and events it signals. Furthermore, the interdependencies of components, i.e. the command and event “wiring,” are also specified in a separate file (.desc). Hierarchical composition of components is also allowed.

The application code as well as the middleware layer must be written using the same model. Before compilation, a preprocessor assembles the necessary system and user-defined components from the appropriate comp and desc files.

5. GRATIS

Working with comp and desc files while developing non-trivial TinyOS applications could quickly become an error-prone and tedious process. Function-like entities (events, commands) have two or more different names in the final application: some on the caller side (e.g. command) and some on the called side (e.g. command handler). This characteristic is inherent in the flexible design enabling the creation of countless different applications without touching the implementation of the individual components. However, as a side effect, it has notable impact on the maintainability of the applications. Introducing a new command or event or renaming an existing one involves the modification of several separate files while having to keep the interface and wiring information in synch.

Even with a simple application, a more expressive representation of the components and the interconnections between them can help design better applications and increase their readability. With more sophisticated components and especially with hierarchical composition this becomes an absolute requirement. The textual representation does not serve this purpose well enough. For example, the information is spread through several files. Visualizing a componentized application while reading multiple text files simultaneously is less than intuitive.

The rules of the wiring are simple; commands and events cannot be connected to each other and the signature (parameter list and return type) of the connected functions on the caller and the called side must be identical. While in most cases these rules are sufficient, some components might impose additional restrictions on their usage. There can be components that are mutually exclusive because they are using the same hardware resources. Some other components can have constraints on the fan-out of their interface points. Unfortunately, these additional requirements cannot be captured using the current methodology.

Model Integrated Computing in general and GME in particular can meet most of these challenges. We have designed a modeling paradigm for GRATIS and configured GME accordingly. The graphical representation provides a solid and intuitive interface for designing and maintaining complex applications. Using this well-defined modeling paradigm the wiring rules and the synchronization between the function names are handled by the modeling environment transparently. The constraint management capabilities of GME allow the specification of the necessary constraints for the components and for the usage of their interfaces.

Our initial goal with GRATIS was to replace the textual representation of the interface and configuration information with a graphical environment, where the desc and comp files are generated automatically and serve just as an intermediate step between the environment and the compiled application. Since all practical applications use system components from the TinyOS distribution, we also had to provide a mapping from the existing large code base to the graphical environment. Therefore, our interpreter not only generates text files from graphical models, but it is also capable of parsing existing files and building the corresponding GME models from them. The main use of this parsing feature is to automatically generate the graphical equivalent of the TinyOS system components and to provide them as a library to the user in the GRATIS environment. We have chosen Python as the implementation language because of its superb text processing capabilities and its solid support for accessing and implementing Microsoft COM objects, the component technology GME is built on.

In any new application domain for GME, such as TinyOS-based application development, important decisions are made during the initial metamodeling process, i.e. the modeling paradigm definition phase. The means by which one captures domain concepts heavily influence the usability and extensibility of the final environment. One might be inclined to map the ideas found in TinyOS directly, namely the separation of application/component wiring (.desc) and the component interface definition (.comp) specifications, where applications contain components and connections while components contain interface points. Instead, we introduced only one modeling concept, the *assembly*. To justify this decision we have to understand how hierarchical decomposition works in TinyOS.

With TinyOS, one can introduce subcomponents in two slightly different ways. In the first scenario a component can be specified having dependencies on other (lower level) components. Using this composite component in any application results in the implicit linking of all required components. One can describe these dependencies by a reference to a desc file in the component description (.comp) using the JOINTLY_ IMPLEMENTED_BY keyword. The referenced description file specifies the required subcomponents and the wiring between them and the original component. Note two important issues here.

One is that the comp file, originally a pure interface description, now contains implementation related information. Second, a desc file can describe not just the whole application but also a composite component.

Moreover, there is yet another type of the hierarchical structure, the proxy-like component, which does not have actual implementation, but provides access points to contained subcomponents. They are defined similarly to composite components, but there is no implementation file and the desc file is referred to using the IMPLEMENTED_BY keyword in the component. These subtle relationships between the comp and desc files lead us to a unified modeling concept, the assembly. The interpretation of an assembly is determined by its content and attributes. Let us summarize these properties and the possible interpretations. The assembly describes an:

- *application*, if it does not contain any interface primitives and does not have a source (.c) file[1], but it does contain references to other assemblies and connections between them (only a desc file will be generated),
- *simple component*, if it does contain access points but does not contain references to other assemblies and it does have a source file (comp and an initial .c files will be generated),
- *composite component*, if it meets the requirements of the simple component, but it contains references to other assemblies (comp, desc and an initial .c files will be generated), or
- *proxy component*, if it meets the requirements of the simple component, but it does not have a source file (comp and desc files will be generated).

Although these rules sound complicated at first, the user does not have to keep them in mind while developing new TinyOS applications. The parser and the code generator apply these rules internally and provide a very intuitive environment for the application designer.

As mentioned earlier the constraint checking capabilities of GME enable us to build a more robust domain-specific environment. The previously described rules are formalized as modeling constraints, and GME guarantees that invalid assemblies cannot be built. Additional constraints restrict the application designer to connect access points only in valid formations.

Both the code generation and the parsing logic are implemented in a single Python file, since they share significant amount of common code (e.g. COM interface to the GME application) and constant values, and because this simplifies the usage and distribution of the GRATIS environment. The parsing process (used to generate a GME library from the TinyOS source tree) can be initiated from the command line and can be controlled by command line parameters (e.g. directory to be parsed, verbose messages, etc.). The parser recursively traverses the specified directory structure while it creates a matching folder hierarchy in a new GME project. In each directory for each comp file it creates a new assembly with the appropriate interface primitives and sets the source file attribute of the assembly if a .c file with the same name can be found. In addition, it records all components that have references to desc files, they will become composite or proxy assemblies. In the second phase it processes these referenced description files, creating references to other assemblies and establishing the connections between them. All the remaining, non-referenced desc files are processed in the last stage and application assemblies are generated.

[1] Assemblies have a boolean attribute representing this information. We set this attribute in the parser if the appropriate .c file is found.

In most cases, we have tried to implement our parsing and wiring logic as close to the original TinyOS implementation as possible. However, we have made one notable modification in the wiring logic. According to the TinyOS implementation, all lines in the desc file containing one or more common interface points are concatenated and handled as one fully connected network. Unfortunately, some topologies cannot be realized with this logic. For example, if one source, A, is connected to multiple destination ports, and another source, B, is connected to only one of them, then B will end up begin connected to all of them anyway. In order to avoid this problem, GRATIS processes all lines in the desc files separately.

In the final phase of parsing, some additional work is performed on the GME project to make the generated library more usable: cleanup and auto placement. During the parsing phases we might encounter problems, inconsistencies in the text files. In these cases the parser tries to handle these errors as gracefully as possible, and it builds the remaining part of the assemblies while marking them. After the parsing process the marked assemblies are deleted along with the empty folders, if the appropriate command line options are set. Otherwise, a warning message is inserted into the erroneous assembly. The auto placement feature moves command handlers and signals to the left side and event handlers and commands to the right side of the assembly, and it also tries to place the contained assembly references evenly. Implementing a more sophisticated heuristic algorithm remains to be done.

An indisputable benefit of the parsing and model building process is an exhaustive testing, since the parser builds and validates all components and applications found in the source tree. In the current TinyOS distribution (0.6) it finds several minor errors that are not caught by the original TinyOS tools.

The code generation procedure is a relatively straightforward process. In this case the Python code is invoked as a COM component from the GME environment. The code generator analyzes the currently opened assembly; it determines what kind of files should be generated according to the rules described earlier, and dumps all the interface and wiring information into the target files. If the assembly has to have an implementation it generates a skeleton .c file if it does not exist already. We are planning to handle existing implementation files in the near future.

Managing changes in the assemblies and maintaining applications across different TinyOS releases are important issues. Since we do not require modifying generated comp and desc files manually, parsing and modifying .c implementation files can resolve the first problem. Although, we have not implemented it yet, this can be achieved with the help of special comments in the implementation files similarly to the technology used in the Visual C++ application and class wizards. To support upgrading applications and components to newer TinyOS releases we need to extend the parser with a special parsing mode, when the comp and desc files are processed with a preloaded GME library, and references are looked up in this library instead of being regenerated.

6. Case study

For demonstrating the benefits of the graphical approach let us study a very simple TinyOS application, one with a single user-defined component and three system components. Note that the real advantages of the visual environment can be recognized more easily with complex applications.

Our sample application, called BLINK, is part of the TinyOS release. It does not provide tremendous functionality: it blinks one of the LEDs at a given frequency. The desc file of the BLINK application specifies the included components and the wiring between the interface primitives.

```
include modules {
   MAIN;
   BLINK;
   CLOCK;
   LEDS;
};

BLINK:BLINK_INIT MAIN:MAIN_SUB_INIT
BLINK:BLINK_START MAIN:MAIN_SUB_START
BLINK:BLINK_LEDy_on LEDS:YELLOW_LED_ON
BLINK:BLINK_LEDy_off LEDS:YELLOW_LED_OFF
BLINK:BLINK_LEDr_on LEDS:RED_LED_ON
BLINK:BLINK_LEDr_off LEDS:RED_LED_OFF
BLINK:BLINK_LEDg_on LEDS:GREEN_LED_ON
BLINK:BLINK_LEDg_off LEDS:GREEN_LED_OFF
BLINK:BLINK_SUB_INIT CLOCK:CLOCK_INIT
BLINK:BLINK_CLK_EVENT CLOCK:CLOCK_FIRE_EVENT
```

Listing 1. BLINK.desc

The MAIN component is contained in every application and its sole role is to initialize and start the application. We have wired these startup functions to our BLINK component. Since our component requires access to the LEDs, we have made the appropriate links the LEDS system component. After initializing the CLOCK component it will generate events periodically. These events are routed to the BLINK component where the event handling routine will toggle the LED.

The interface of our BLINK component can be easily defined now. From the wiring we know what kind of commands and events must be handled in our component and what other commands will be used by it. Unfortunately, the signature of these interface primitives must be looked up from the comp files in the TinyOS distribution.

The graphical equivalent of the previous configuration files is shown in Figure 4. It contains four references to four assemblies, three of them are dragged into the application from the TinyOS library and the fourth one

(BLINK) was implemented as a user-defined assembly. The visual representation of the wiring is undeniably easier to understand, for example, the directions of the links are unambiguous to the application designer. If later we decide to rename one of our functions, we will have to modify it in one place only, in the proper assembly.

```
TOS_MODULE BLINK;
ACCEPTS {
    char BLINK_INIT(void);
    char BLINK_START(void);
};

HANDLES {
    void BLINK_CLK_EVENT(void);
};

USES {
    char BLINK_SUB_INIT(char ival, char sc);
    char BLINK_LEDr_on();
    char BLINK_LEDr_off();
    char BLINK_LEDy_on();
    char BLINK_LEDy_off();
    char BLINK_LEDg_on();
    char BLINK_LEDg_off();
};

SIGNALS {
};
```

Listing 2. BLINK.comp

Because the TinyOS distribution contains the BLINK application as an example, the model shown in Figure 4 was not built by hand, but generated by our parser tool.

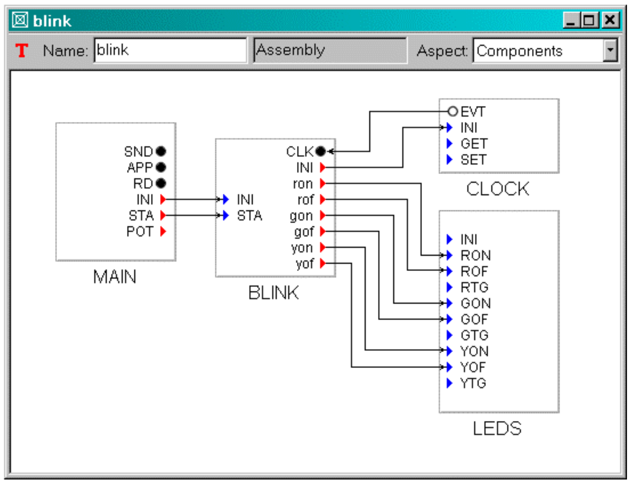
Name: blink
Assembly
Aspect: Components

Figure 4. BLINK assembly

7. Conclusions

There are two important points we would like to emphasize. First, we claim that domain-specific modeling and code generation environments are a natural fit for component-based software development. Second, even though the development of these environments from scratch is expensive, our solution of applying a metaprogrammable environment makes them quite feasible even for relatively narrow domains such as TinyOS-based embedded system development.

GRATIS is a fully functional sophisticated environment for TinyOS application development. The effort to create the toolset was exactly ***one man-month*** excluding an initial week of learning the ins and outs of TinyOS, since we never used it before. Furthermore, 75% percent of the effort was the development of the Python parsing code that builds up the model library from the TinyOS source. This clearly demonstrates the significant productivity increase in tool development that can be achieved with Model Integrated Computing in general and GME in particular. Note, however, that we are expert users of our own tools and their learning curve is quite steep, so a novice GME user cannot expect this level of productivity initially. However, experience shows that after a relatively short training period, users can be become quite productive very rapidly.

On the actual domain-specific tool side, there is no experimental data yet on how much productivity increase can be achieved by GRATIS over traditional text-based development. However, the features of no redundant information specification, error checking, constraint enforcement, increased readability of the visual representation, assured consistency across components and automatic code generation are undoubtedly significant advantages beyond the expected measurable productivity gain.

Both GME and GRATIS are freely available from http://www.isis.vanderbilt.edu/projects/ in the gme and nest subdirectories respectively.

8. Acknowledgement

The DARPA/ITO NEST program (F33615-01-C-1903) has supported, in part, the activities described in this paper.

9. References

[1] J. Sztipanovits, G. Karsai, "Model-Integrated Computing," *Computer*, April, 1997. pp. 110-112
[2] A. Ledeczi et al.: "Composing Domain-Specific Design Environments," *Computer*, November, 2001. pp. 44-51
[3] J. Rumbaugh, I. Jacobson, G. Booch: "*The Unified Modeling Language Reference Manual*," Addison-Wesley, 1998
[4] J. B. Warmer, A. G. Kleppe: "*The Object Constraint Language : Precise Modeling With Uml*," Addison-Wesley, March 1999
[5] J. Hill et al.: "System Architecture Directions for Networked Sensors," Proceedings of ASPLOS, 2000