Using BIP for Modeling and Verification of Networked Systems – A Case Study on TinyOS-based Networks

Ananda Basu, Laurent Mounier, Marc Poulhiès, Jacques Pulou, Joseph Sifakis
{basu, mounier, poulhies, sifakis}@imag.fr, jacques.pulou@orange-ftgroup.com

Abstract

We apply a model construction methodology to TinyOS-based networks, using the Behavior-Interaction-Priority *(BIP) component framework. The methodology consists in building the model of a node as the composition of a model extracted from a nesC program describing the application, and models of TinyOS components. Models for networks are obtained by composition of models for nodes by using BIP connectors implementing different types of radio channels. This opens the way for enhanced analysis and early error detection by using verification techniques.*

1. Introduction

Wireless sensor networks are complex component-based systems with rich dynamics subject to strong extra-functional requirements. Their design involves the composition of a variety of hardware and software components developed with different methodologies and tools. We have a limited understanding on how specific component features impact the global behavior. To cope with complexity and enhance understanding, it is important to consider wireless sensor networks as the composition of a relatively small set of functions, services and components by using incremental structuring principles. The main obstacle for this is the lack of modeling frameworks encompassing heterogeneity. Most simulation environments use simulation software built in a more or less ad hoc manner, by integrating the application code in specific platforms [7, 6, 10, 8, 5]. They can be useful for debugging purposes but they are not adequate for a more thorough exploration of a network's non-deterministic dynamics.

We apply to TinyOS-based networks, a model construction methodology for building heterogeneous real-time systems. This opens the way for enhanced analysis and early error detection by using verifications techniques. The methodology is not specific to TinyOS, and we believe, can be adapted to networked systems, in general. It uses the *Behavior-Interaction-Priority* (BIP) component framework [3]. BIP consists of a language for modeling component-based systems and associated execution/simulation and verification tools. It has sound theoretical foundations based on operational semantics implemented by a dedicated execution/simulation platform.

For a given sensor node, a global BIP model is built by composing BIP models for its application software and for TinyOS. The latter is obtained by composing controllers for the execution of tasks, events, radio and hardware devices. The models for application software are generated automatically from nesC programs by a translator (shown in figure 1) which takes annotated nesC code as input and generates the corresponding BIP components and connectors. BIP models can be analyzed by using powerful state space exploration techniques offered by the IF toolset [4].

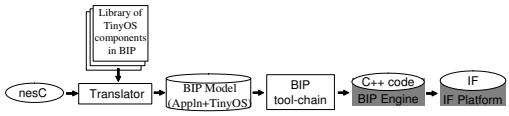

Figure 1. The modeling flow.

The methodology presented is characterized as follows:

• A global model for the network is built by composition of BIP components modeling the application software as well as operating system and radio features. This is a main difference with existing simulation approaches, directly using TinyOS and C code generated by the nesC compiler. The BIP model for the TinyOS is an abstract machine driving the execution of the BIP model, obtained by translation of the application software written in nesC.

• A significant difference with existing simulation approaches, is that the obtained BIP models are non-deterministic and fully characterize the behavior of the wireless sensor network, independent of the used platform. Furthermore, these models have a well-defined notion of state. They can be verified by using state space exploration techniques e.g., model-checking.

• Another important difference is incremental model construction of BIP models [9]. Incrementality means that the global model is obtained by progressively composing its

atomic components. This allows preservation of the structure through translation into BIP. That is, it is possible to identify in the global model all its atomic components and their interactions. This allows in particular, to study the impact of changes of a component's behavior or structure on the global behavior and its properties.

The paper makes the following three main contributions.

• It provides a methodology for building global and faithful models for heterogeneous networked systems.

• It allows a better understanding of the interplay between platform-dependent and platform-independent features. The model of a node is the composition of an abstract machine modeling TinyOS, and a system-oriented model of its application software.

• It provides a single framework supporting both behavioral verification and simulation of networked systems. A comparison on common benchmarks with state-of-the-art simulation environments, shows that this is possible without significant performance degradation.

We assume the reader is familiar with nesC. Informations about BIP can be found in [3]. An extended version of this paper can be found in a technical report [1].

The paper is structured as follows. Section 2 describes the modeling principle for nesC programs. Section 3 describes the modeling principle for TinyOS. The global model construction is explained in Section 4, as the composition between application and TinyOS components. We present experimental results for three examples in Section 5 and conclude in Section 6.

2. Modeling user-defined nesC components

We use a translator that takes annotated non re-entrant nesC code as input and generates the corresponding BIP components and connectors. Annotations are used to extract the structure characterized by the set of atomic components and the connectors between them.

The method consists in transforming implementations of the Commands, Events and Tasks in a nesC program into atomic BIP components representing Command handlers, Event handlers, and Task handlers, respectively. The modeling of the behavior of the atomic components is left to the user. The non re-entrancy limitation can be overcome by using richer models in BIP. It is possible to detect re-entrancy in BIP models by using verification tools.

A generic BIP model for atomic components is shown in figure 2. The interface consists of a set of ports with associated types. The behavior is specified by the control states *IDLE*, *SUSP* and *EXE* with transitions between them labeled by ports corresponding to respective actions. *EXE* is a macro state and is further decomposed into states and transitions depending on the specific behavior of the particular component.

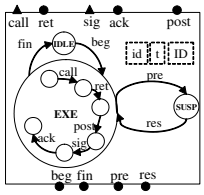

Figure 2. A nesC module in BIP.

The ports are classified in two groups:

• The first consists of the ports *beg, fin, pre* and *res* labeling the transitions for beginning, finishing, preempting and resuming execution of a component. These ports may be used in interactions between the component and TinyOS or in interactions implementing call/return mechanisms for Command handlers. They are *incomplete* [3] as they require triggering from other components.

• The second consists of the ports *call, ret, sig, ack, post* labeling the transitions for call and return of commands, signaling and acknowledgment of events and posting of tasks. The ports *call* and *sig* are of type *complete* [3] as they are triggers of broadcast connectors.

A generated component also contains, in addition to specific local variables, generic variables representing its unique identifier (*ID*), the identifier of a callee (*id*) and the identifier of a posted Task (*t*).

3. Modeling TinyOS in BIP

Our TinyOS model is the composition of two sets of components: 1) schedulers for Events and Tasks, 2) models for hardware components representing Timers, Sensors and Radio.

• Modeling Scheduler : We use two schedulers to model the two-level scheduling mechanism of TinyOS. The *Event Scheduler* (figure 3(a)) is responsible for the management of events generated by hardware components. When a hardware-generated event *e* is received through the port `sig`, the scheduler first preempts any running component by synchronizing through the port `pre` and stacks the id's of the preempted components received . Then, it triggers the execution of the Event handlers identified by *e* by broadcasting *e* through the port `beg`. From state *BUSY1*, the *Event Scheduler* can either be triggered by a new hardware-generated signal (port `sig`), or by a finish notification (port `fin`). In the first case, it preempts the currently running component, in the second case, depending on the state of the stack (empty or not), it goes to *IDLE* or to *BUSY2* from which it resumes the last preempted component.

The *Task Scheduler* (figure 3(b)) receive new task postings through its port *post* and treats the tasks in FIFO order. It can start a new task only if the *Event Scheduler* is *IDLE*.

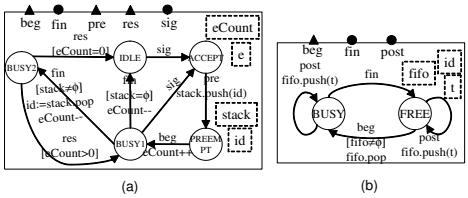

Figure 3. Event(a) and Task(b) Schedulers.

• Timers, Sensors and Radio controllers are modeled as BIP components in a similar way.

4. The global architecture

In this section we describe the composition of the BIP components using connectors, to build the model of a node as well as the model of the network by specifying interactions between the nodes.

1) Interactions within a node : We explain the principles of construction of BIP model for nodes by using two sets of connectors.

• The first set models interactions between nesC components for *call* statements and *signal* statements issued by software. A typical *call* statement will generate a *Call* connector and a set of $Return_i$ connectors as shown in figure 4. The *Call* connector is a *broadcast* connecting the *call* port of the caller (c) to the *beg* ports of the possible callees (p, q, r). The component c may call either p and q jointly leading to the interaction (`c.call, p.beg, q.beg`), or call r leading to the interaction (`c.call, r.beg`). The selection of one of these interactions is by using activation conditions involving comparisons between callee identifiers (ID) and the calling identifier (id) (not shown in figure 4).

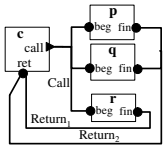

Figure 4. Connectors for a nesC call.

The $Return_i$ connectors synchronize the *fin* ports of the callees to the *ret* port of the caller.

The *signal* statements representing software event signalling are handled exactly in the same manner as the *call* statements explained above. However, signals representing hardware events are treated separately and are processed by the event scheduler.

• The second set of connectors deal with interactions between BIP components for the application and BIP components for TinyOS (see figure 5). The connectors *TBegin* and

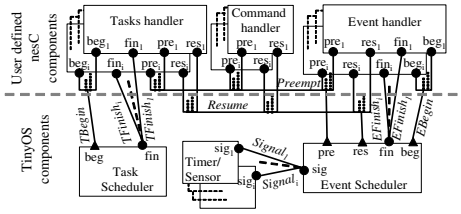

Figure 5. The global architecture in BIP.

EBegin deal respectively with interactions between Tasks handlers/Task Scheduler and Event handlers/Event Scheduler. The connectors $TFinish_i$ and $EFinish_i$ are used by Tasks and Event handlers to notify their completion. The *Preempt* connector triggers preemption of the application components. The *Resume* connector is used to resume execution of the last suspended component. The connectors $Signal_i$ are used to signal any hardware-generated events.

Task posting is through connectors between the port *post* of the Task Scheduler and the ports *post* of software components (not shown in the figure).

2) Modeling Radio Links : Radio links are modelled as BIP connectors linking the BIP components modelling radio controllers. These components have a *broadcast* port to send data and a *listen* port to receive data. We consider networks with static topology and use only one connector per *broadcast* port. This connector links the *broadcast* port with all the receivers, through their *listen* port. For each connector, activation conditions depending on the distance between sender and receiver are used to define the feasible interactions. More complex activation conditions allow modelling lossy links.

5. Experimental results

We consider 3 examples: *BlinkTask*, *SenseToLeds* and *SenderReceiver*.

The first example illustrates the utilization of verification techniques. The two others compare our method to specific state-of-the-art simulation methods. One would expect that the use of a general purpose modeling technique instead of a specific one, well-tuned for a particular execution platform, would have a strongly negative impact on performance. Furthermore, the use of rich (non-deterministic) models instead of deterministic ones, could also have a similar effect. Experimental results show no significant performance degradation in comparison with [5] for example.

BlinkTask[2] describes a node with a variable *state* representing the state of its LED. This variable is shared between the Task *processing*, which reads it, and the Event handler *Timer.fired()*, which modifies it. For *BlinkTask* we generated a timed BIP model with 4 user-defined atomic components, 3 TinyOS components (2 schedulers and 1

Timer) and 11 connectors. Exhaustive state space exploration allows detecting error states where a new timer interrupt arrives while the Task *processing* is still being executed. Traces leading to such error states can be obtained by modeling an *Observer* component in BIP, keeping track of the sequence of interactions of the node. As an example, the analyzed state graph has 28,701 states and 46,197 transitions for the following execution time intervals: *Timer* period $[50, 50]$, *Timer.fired()* $[2, 9]$, *Leds.redOn()* $[2, 7]$, *Leds.redOff()* $[2, 7]$, *processing()* $[20, 32]$. The selected values ensure a correct behavior of the example. However, changing the timer period to values less than $[48, 48]$ leads to error states as detected by the *Observer*.

The second example is *SenseToLeds*[2] which is a node sampling data from a photo Sensor and displaying them in the LEDs. Its nesC code consists of 4 components. The translation to BIP produces 8 user-defined components, 4 TinyOS components (2 schedulers, 1 Timer and 1 Sensor), and 21 connectors.

We consider a network of 250 *SenseToLeds* nodes without radio links. For a virtual run time of 300 seconds, considering a 4 Hz timer on each node for the network, the simulation takes 600 seconds on a standard desktop computer, which stays reasonable. As expected, the simulation time increases linearly with the number of nodes.

The third example *SenderReceiver* is a network of senders and receivers, with lossless channels and static topology. Each sender is connected to a fixed number of receivers y. Each receiver has a unique sender (no collision). The sender nodes execute the *CntToLedsAndRfm*[2] nesC program, and the receiver nodes execute the *RfmToLeds*[2] program. Figure 6 shows real execution times for 300 virtual seconds considering a 4 Hz timer on each node, as a function of the number of senders x and the number of receivers per sender y.

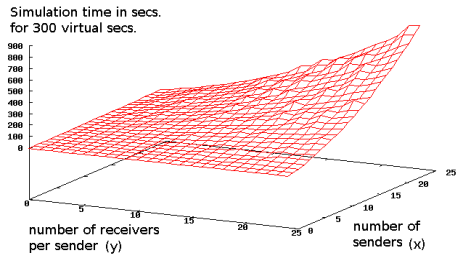

Figure 6. *SenderReceiver* **example.**

6. Conclusion

The paper applies to TinyOS, a methodology for modeling and verification of networked systems. The methodology is general and can be applied to building global models of heterogeneous systems. It consists in modeling the execution platform as an abstract machine driving the execution of the application software. For this, a formalization of the language in which application software is written must be provided, in terms of the primitives offered by the platform. This is certainly not an easy task. The formalization should be made at the right abstraction level. Computational granularity should be chosen so as to include in the model all the events which are relevant for the properties to be verified. Furthermore, to keep model complexity low, it should ignore computation sequences not involving such events. The model generation methodology applied to nesC, can be adapted to any language used for programming applications. Its parser can be adequately engineered to identify in the source code, constructs generating relevant events and determine computation granularity. This can be used for (compositionally) generating BIP code.

We spent two man-months for developing the methodology for TinyOS. For other platforms, much more effort would be needed for feature componentization at the right abstraction level. Such an investment seems to be the only way to overcome current limitations of model-based design and to design systems of guaranteed quality.

References

[1] http://www-verimag.imag.fr/index.php?page=techrep-list.

[2] http://www.tinyos.net/.

[3] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-Time Components in BIP. In *SEFM06, IEEE Computer Society*.

[4] M. Bozga, S. Graf, and L. Mounier. IF-2.0: A Validation Environment for Component-Based Real-Time Systems. CAV02.

[5] E. A. L. Elaine Cheong and Y. Zhao. Joint modeling and design of wireless networks and sensor node software. Technical Report UCB/EECS-2006-150, EECS Department, University of California, Berkeley, November 2006.

[6] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation and deployement of heterogeneous sensor networks. In *2nd International Conference on Embedded Networked Sensor Systems*. ACM Press, 2004.

[7] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire TinyOS applications. In *SenSys '03: 1st International Conference on Embedded networked sensor systems*, pages 126–137. ACM Press.

[8] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. S. Baras. ATEMU: A Fine-grained Sensor Network Simulator. In *Proceedings of SECON*, 2004.

[9] J. Sifakis. A framework for component-based construction. In *SEFM05, pages 293-300*, pages 293–300. IEEE Computer Society.

[10] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. In *IPSN 05*, 2005.