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

Complex heterogeneous systems such as networked systems, composed of hardware and software, are validated by simulation of physical or virtual prototypes. The main obstacle for the application of verification techniques, which are successfully applied to complex software or hardware, is the lack of methods for building global models faithfully representing their behavior.

We apply a model construction methodology using the Behavior-Interaction-Priority *(BIP) component framework, to TinyOS-based networks. 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 connectors implementing different types of radio channels. This opens the way for enhanced analysis and early error detection by using verification techniques.*

1 Introduction

Modeling and verification techniques have been successfully applied to complex software or hardware. Currently, validation of complex heterogeneous systems such as networked systems, is carried out by simulation or testing of prototype implementations. Existing verification techniques could be applied to heterogeneous systems, provided that we have methods for building executable models faithfully representing their behavior. The construction of such models by composition of models of the application software and of the underlying execution platform is a scientific and technical challenge.

A main difficulty for jointly modeling an application software and its execution infrastructure, is that they adopt very different execution models and views. In component-based software, components are mainly used for structuring functions and associated data. Interactions between components are point-to-point (*e.g.* function calls) through binding interface specifications. This view is far from a system-oriented view needed to model execution mechanisms and their interaction with the external environment. For instance, programs in the nesC language used for programming TinyOS-based applications [6], are sets of components and relations between *provided* and *used* interfaces. This programmer's view is not sufficient for determining the interactions between the application software and TinyOS which manages entities such as tasks, commands and events by applying specific scheduling rules.

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 [8, 7, 11, 9, 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 [2].

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, 3].

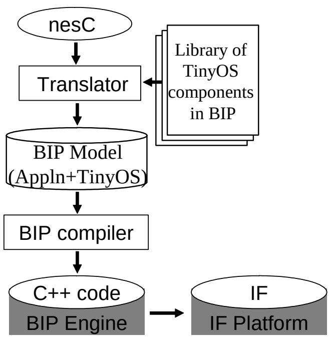


Figure 1. The modeling flow.

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. Furthermore, these models have a well-defined notion of state. They can be verified by using state space exploration techniques e.g., model-checking. Even if due to inherent limitations, complete verification of complex networks is intractable, verification is very useful for systematic debugging and early error detection.

• Another important difference is incremental model construction of BIP models [10]. 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.

The paper is structured as follows. Section 2 provides a succinct presentation of BIP, the underlying modeling methodology and supporting tools. An informal presentation of nesC and its semantics is given in Section 3. Section 4 describes the modeling principle for nesC programs. Section 5 describes the modeling principle for TinyOS. The global model construction is explained in Section 6, as the composition between application and TinyOS components. We present experimental results for three examples in Section 7 and conclude in Section 8.

2 The BIP component framework

BIP[1][2] is a software framework for modeling heterogeneous real-time components. The BIP component model is the superposition of three layers: the lower layer describes the *behavior* of a component

[1]BIP stands for *Behavior, Interaction, Priority* and can be downloaded:
`http://www-verimag.imag.fr/~async/BIP/bip.html`.

as a set of *transitions* (*i.e* a finite state automaton extended with data); the intermediate layer includes *connectors* describing the *interactions* between transitions of the layer underneath; the upper layer consists of a set of *priority* rules used to describe scheduling policies for interactions. Such a layering offers a clear separation between component behavior and structure of a system (interactions and priorities).

The BIP framework consists of a language and a toolset including a frontend for editing and parsing BIP programs and a dedicated platform for the model validation. The platform consists of an Engine and software infrastructure for executing models. It allows state space exploration and provides access to model-checking tools of the IF toolset [4, 3]. This permits to validate BIP models and ensure that they meet properties such as deadlock-freedom, state invariants and schedulability.

The BIP language allows hierarchical construction of *compound components* from *atomic* ones by using connectors and priorities.

An *atomic* component consists of a set of *ports* used for the synchronization with other components, a set of transitions and a set of local variables. Transitions describe the behavior of the component. They are represented as a labeled relation between *control states*. A transition is labeled with a port p, a guard g and a function f written in C. The guard g is a boolean expression on local variables and the function is a block of C code. When g is true, f is executed if an interaction involving p occurs.

Interactions between components are specified by *connectors*. A connector is a list of ports of atomic components which may interact. For instance, `(task1.call, task2.begin, task3.begin)` is a connector relating respectively the ports `call`, `begin`, `begin` of instances `task1`, `task2`, `task3` of a generic component `Task`, as shown in figure 2(a). To determine the interactions of a connector, its ports have the synchronization attributes *complete* or *incomplete*, represented graphically by a triangle and a bullet, respectively. A connector defines a set of interactions defined by the following rules:

• If all the ports of a connector are incomplete then synchronization is by *rendezvous*. That is, only one interaction is possible, the interaction including all the ports of the connector.

• If a connector has one complete port then synchronization is by *broadcast*. That is, the complete port may synchronize with the other ports of the connector. The possible interactions are the non empty sublists containing this complete port.

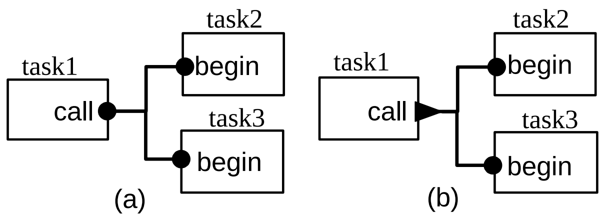


Figure 2. BIP port types and connectors.

In figure 2(a), all the ports are incomplete, so the only feasible interaction is the rendezvous `(task1.call, task2.begin, task3.begin)`.

In figure 2(b), as `call` is complete and `begin` ports are incomplete, the feasible interactions are (`task1.call`), (`task1.call`, `task2.begin`), (`task1.call`, `task3.begin`) and (`task1.call`, `task2.begin`, `task3.begin`).

In BIP, it is possible to associate with an interaction an activation condition (guard) and a data transfer function both written in C. The interaction is possible if components are ready to communicate through its ports and its activation condition is true. Its execution starts with the computation of data transfer function followed by notification of its completion to the interacting components.

3 The nesC programming model – informal semantics

We briefly present nesC, an extension of C used to develop TinyOS applications [6].

nesC applications are built by writing and assembling *components*, representing either software (e.g., a protocol layer) or hardware (e.g., radio devices, timers, sensors). Components *provide* and *use* interfaces, which are groups of services. Interfaces contain *commands* and *events*.

The providers of an interface implement the commands (by means of *command handlers*), while the users implement the events (by means of *event handlers*). This distinction between commands and events within the same interface, allows to properly implement the so-called *split phase* mechanism: the execution of a non atomic operation (e.g., sending a packet) is split into two distinct phases, a command call to request the operation, and an event reception indicating its termination.

It is also possible to use deferred computation mechanisms called *tasks*. A nesC application is therefore written in C code, extended with a few extra primitives, i.e., *call* a command, *signal* an event, and *post* a task.

There are two types of components in nesC: *modules* and *configurations*. Modules provide application code, implementing one or more interfaces. Configurations are used to wire components together. Note that the wiring relation between components is not point to point. In particular, a command call performed by a component can be bound to several Command handlers provided by other components. After a call, the caller waits for completion of *all* the activated callees. Return values are then merged by using a combination function. Event signaling by software components is handled in a similar manner.

Execution of nesC applications is handled by a two-level TinyOS scheduler.

The first level manages task execution, for background computations. The TinyOS scheduler follows a strict FIFO policy for tasks: pending tasks are stored in a FIFO queue, and a task cannot be preempted by another task. Posting a task is a non-blocking operation that returns immediately. A return value indicates either a successful or an unsuccessful post operation (e.g., when the task queue is full).

The second scheduling level is used for event execution. Events represent either hardware interrupts, or indicate the completion of a given requested service. Execution of an event handler is *preemptive*: when an event is received, its corresponding event handler(s) is/are immediately activated, interrupting the current computation (which could be either a task, or another event handler). The suspended execution will resume at the end of event handler execution. Note that this policy may lead to code re-entrance (e.g., when an instance of an event handler preempts another instance of the same event handler).

Sections 4, 5, 6 present three steps for the construction of a global sensor network model in BIP: 1) generation of BIP components from user-defined nesC components, 2) instantiation of predefined BIP components modeling TinyOS, radio and sensors and 3) composition of these components by using connectors modeling communication links.

4 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 modeling of the behavior of the atomic components is left to the user.

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 non re-entrancy limitation can be overcome by using richer models in BIP. It is possible to detect re-entrance in BIP models by using verification tools.

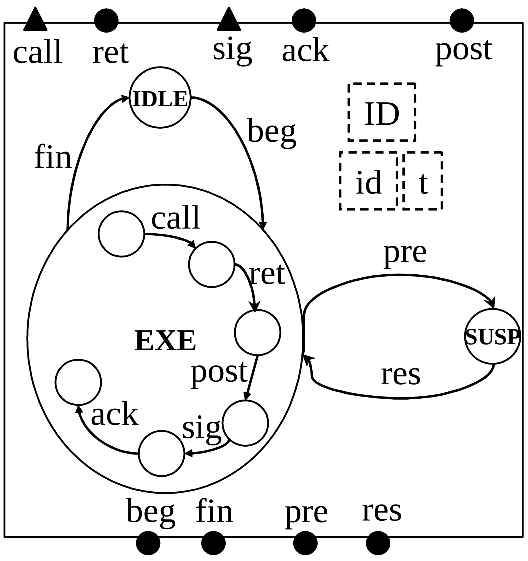


Figure 3. A nesC module in BIP.

A generic BIP model for atomic components is shown in figure 3. 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.

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* 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* 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*).

5 Modeling TinyOS in BIP

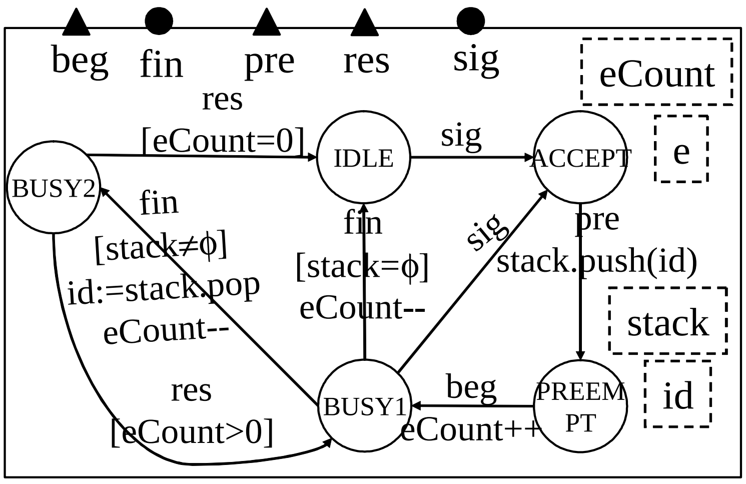
Our TinyOS model is the composition of two sets of components: 1) schedulers for Events and Tasks, 2) models for hardware components representing Timers, Sensors and Radio.

5.1 Scheduler modeling

We use two schedulers to model the two-level scheduling mechanism of TinyOS.

The *Event Scheduler* (figure 4(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 4(b)) is responsible for the scheduling of tasks. It treats the tasks in FIFO order and waits for a task to finish before starting a new one. It has two states: *FREE* and *BUSY*, depending on whether a task is executing or not. In any of these states, it can synchronize through its



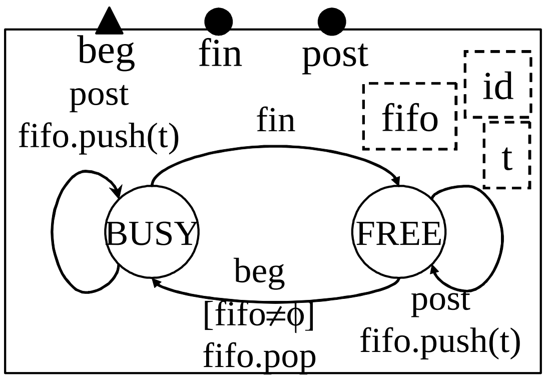


Figure 4. Event(a) and Task(b) Schedulers.

port *post* to receive new task postings. In the *BUSY* state, it waits for the currently executing task to finish and goes back to the *FREE* state. It can start a new task only if the *Event Scheduler* is *IDLE*.

5.2 Hardware modeling

5.2.1 Radio Controller

Each node has a radio controller composed of a *Radio Sender* (figure 5(a)) and a *Radio Receiver* (figure 5(b)). We consider a packet level radio model where packet sending is an atomic operation. Sending a packet is a split-phase mechanism modeled by the Command handler *send* and the Event handler *sendDone*. The *send* Command handler is called from the application, and is a request to send a packet through the radio. It synchronizes with the *Radio Sender* through the *sync_send* port which passes the packet to the *Radio Sender*. Then, the *Radio Sender* broadcasts the packet. This is followed by triggering the Event handler *sendDone*.

The *Radio Receiver* receives a packet through the *listen* port, and then, it triggers the Event handler *receive*.

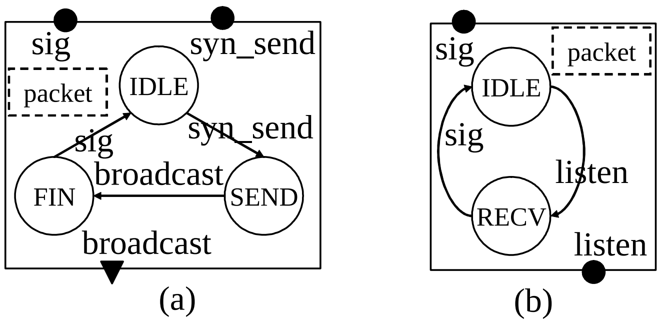


Figure 5. Radio controller components.

5.2.2 Timers and sensors

A Timer component is a simple BIP component with a single state and two transitions. One transition is labeled by port *sig* to signal an expiration event. The other is labeled by a special port *tick* and is used to count time steps. To ensure time consistency, the *tick* ports of all the Timers are incomplete and strongly synchronized by using a single connector.

In nesC, Sensors are hardware modules offering interfaces for split-phase operation. The BIP description consists of a model for the Sensor itself, along with the models for the Command handler *getData* and Event handler *dataReady*. The actual value read by the Sensor component can be either a random value or a value provided by a model of the environment. The latter can also be explicitly modeled in BIP.

6 Modeling interaction between the components - 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.

6.1 Interactions in a node

We explain the principles of construction of BIP model for nodes by using two sets of connectors.

The first set models interactions 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 6.

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 interactions is by using activation conditions involving comparisons between callee identifiers (ID) and the calling identifier (id).

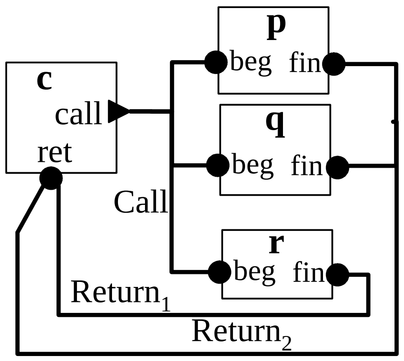


Figure 6. BIP connectors for a nesC call command.

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.

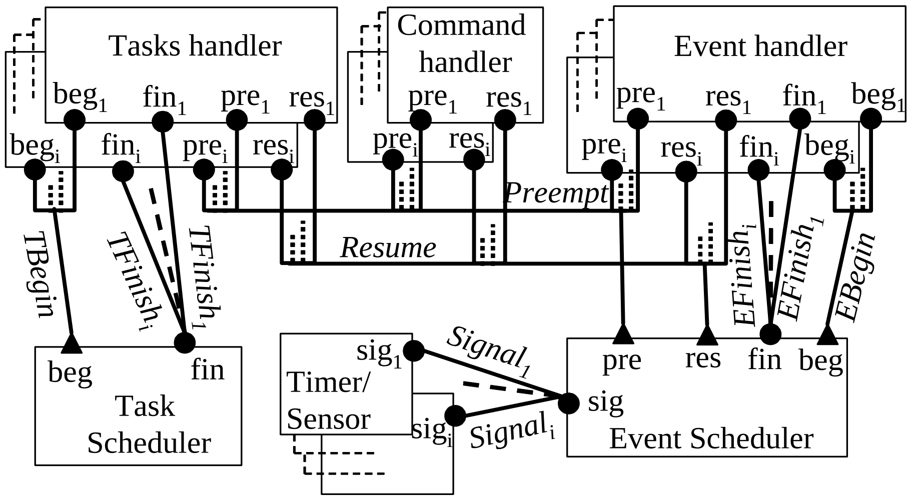


Figure 7. The global architecture in BIP.

The second set of connectors deal with interactions between BIP components for the application and BIP components for TinyOS (see figure 7).

The connectors *TBegin* and *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).

6.2 Interactions between nodes - Radio Links

Radio links are modelled as BIP connectors linking the ports *broadcast* and *listen* of the radio controller. 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.

7 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.

BlinkTask[1] 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*[1] 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 *SenseToLeds* nodes without radio links. We show in figure 8, simulation times as a function of the number of nodes for a virtual run time of 300 seconds, considering a 4 Hz timer on each node. We performed the tests on an AMD Athlon XP 2800+, 1Gb of RAM running GNU/Linux. The execution time for the network increased linearly with the number of nodes, as expected.

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*[1] nesC program, and the receiver nodes execute the *RfmToLeds*[1] program. Figure 9 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.

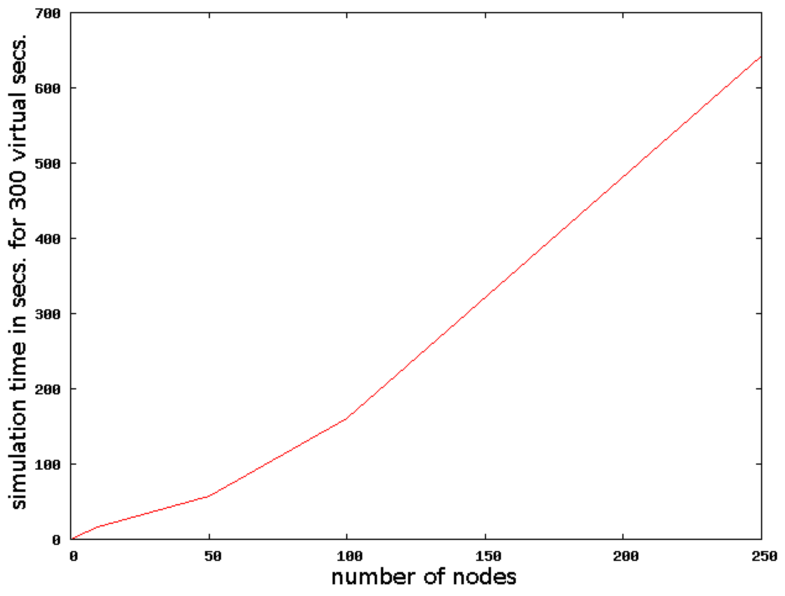


Figure 8. *SenseToLeds* **example.**

8 Conclusion

Currently, validation of complex heterogeneous systems: such as networked systems, is carried out by simulation or testing of prototype implementations. Verification techniques such as model-checking and static analysis are already successfully used for software or hardware. They could be extended to heterogeneous systems, provided that we have methods for building executable models for these systems.

The paper applies to TinyOS, a methodology for modeling and verification of networked systems. The methodology is based on the use of the BIP component framework which encompasses description of heterogeneous real-time systems. It allows the construction of global models obtained as the composition of models of nodes. These are obtained by composition of models of the application software and of the execution platform.

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. Computation granularity should be chosen

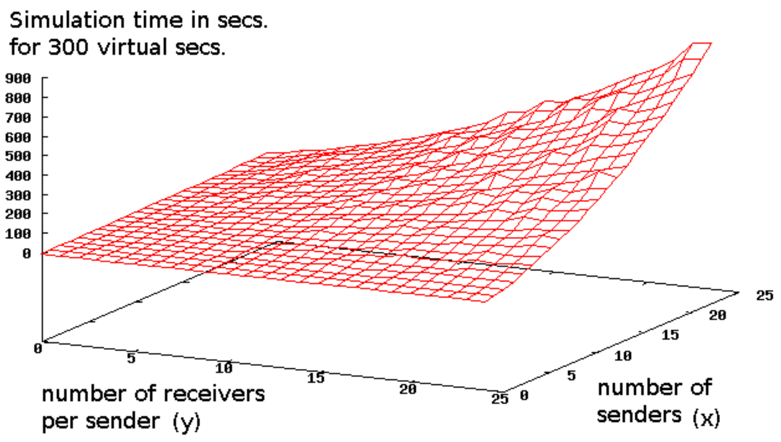


Figure 9. *SenderReceiver* **example.**

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. For instance, for the verification of synchronization and resource properties, it should assemble atomic sequences of code. 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$\times$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 for overcoming current limitations of model-based design and for designing systems of guaranteed quality.

Currently, behavioral aspects of networks are validated using specific simulation environments built in some ad hoc manner and integrating application code, protocols and platforms. Our approach allows the use of a single modeling framework supporting a disciplined system construction methodology. It allows the systematic construction of global models spanning all possible system execution sequences. The results show that using such a non-specific framework and rich models does not entail significant performance overhead. The advantages are numerous, including enhanced analysis and verification as well as comparison of implementations of the same application on different platforms.

References

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

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

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

[4] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF Toolset. In *School on Formal Methods for the Design of Computer, Communication and Software Systems*, September 2004.

[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] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[7] 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.

[8] 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.

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

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

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