Proceedings of the International Multiconference on Computer Science and Information Technology pp. 561–568

ISSN 1896-7094
© 2006 PIPS

From Simulation to Execution: on a Certain Programming Paradigm for Reactive Systems

Wlodek Dobosiewicz[1] and Pawel Gburzynski[2]

[1] University of Guelph, Department of Computing and Information Science, Guelph, Ontario, CANADA N1G 2W1
[2] University of Alberta, Department of Computing Science, Edmonton, Alberta, CANADA T6G 2E8

Abstract. We present a certain programming environment, which (over the years of its evolution) has allowed us to unify simulation, specification and execution for a class of highly practical systems, including (low-level) communication protocols and embedded software. The programming paradigm inherent in our scheme is a combination of FSM (finite state machines) and coroutines with multiple entry points. It brings about a unique flavor of multitasking suitable and appropriate for expressing both event driven simulation models as well as executable, multithreaded, embedded systems. Its characteristic feature in the latter domain is the extremely small footprint, which makes it possible to program tiny devices in a highly structural, reusable, self-documenting and reliable manner.

1 Introduction

The project that has resulted in the set of tools discussed here started in 1986 as study of collision-based (CSMA/CD) medium access control (MAC) protocols for wired (local-area) networks. About that time, many publicized performance aspects of CSMA/CD-based networks (like Ethernet) had been subjected to heavy criticism from the more practically inclined members of the community—for their apparent irrelevance and overly pessimistic confusing conclusions. The culprit, or rather culprits, were identified among the popular collection of models (both analytical and simulation), whose cavalier application to describing poorly understood and crudely approximated phenomena had resulted in worthless numbers and exaggerated blanket claims [4]. Our own studies of low-level protocols for local-area networks, on which we embarked at that time [6, 10, 14, 7, 8, 13, 9, 12], were aimed at devising novel solutions, as well as dispelling myths surrounding the old ones. Owing to the fact that exact analytical models of the interesting systems were (and still are) nowhere in sight, the performance evaluation component of our work relied heavily on simulation. To that end, we developed a detailed network simulator, called LANSF,[3] and its successor SMURPH,[4] [15, 11], which carefully accounted for all the relevant physical phenomena affecting the correctness and performance of low-level protocols, e.g., the finite propagation speed of signals, race conditions, imperfect synchronization of clocks, variable event latency incurred by realistic hardware. In addition to numerous tools facilitating performance studies, SMURPH was also equipped with instruments for dynamic conformance testing [3] (see [20] for another implementation of this idea).

At some point we couldn't help noticing that the close-to-implementation appearance of SMURPH models went beyond mere simulation: the same paradigm might be useful for implementing certain types of real-life applications. The first outcome of this observation was an extension of SMURPH into a programming platform for building distributed controllers of physical equipment represented by collections of sensors and actuators. Under its new name, SIDE,[5] the package encompassed the old simulator augmented by tools for interfacing its programs to real-life objects [16, 17].

A natural next step was to build a complete and self-sustained executable platform (i.e., an operating system) based entirely on SMURPH. It was directly inspired by a highly practical project whose objective was to develop a credit-card-sized device equipped with a low-bandwidth, short-range, wireless transceiver allowing it to communicate with neighbors. As most of the complexity of the device's behavior was in the communication protocol, its model was implemented and verified in SMURPH.

[3] Local Area Network Simulation Facility.
[4] A System for Modeling Unslotted Real-time PHenomena.
[5] Sensors In a Distributed Environment.

The source code of the model, along with its plain-language description, was then sent to the manufacturer for a physical implementation. Some time later, the manufacturer sent us back their prototype microprogram for "optical" conformance assessment. Striving to fit the program's resources into as little memory (RAM) as possible, the implementer organized it as an extremely messy single thread for the bare CPU. The thread tried to approximate the behavior of our high-level multi-threaded model via an unintelligible combination of flags, hardware timers and counters. Its original, clear, and self-documenting structure, consisting of a handful of simple threads presented as finite state machines, had completely disappeared in the process of implementation. While struggling to comprehend the implementation, we designed PicOS: a tiny operating system providing for an easy, natural and rigorous implementation of SMURPH models on microcontrollers. Even to our surprise, we were able to actually *reduce* the RAM requirements of the re-programmed application. Needless to say, the implementation became clean and clear: its verification was immediate.

As of the time of writing, PicOS is gaining a foothold in the commercial world—being used as a basis for rapid prototyping and implementation of software for tiny wireless devices.[6] It can be viewed as an alternative to TinyOS [18], with several practical advantages over that system.

The most recent step (work in progress) in the evolution of LANSF, SMURPH, SIDE and PicOS stems from the practical applications of PicOS in small-footprint wireless systems, and, in a way, closes the circle. The large number of generic applications for the wireless devices, combined with the obvious limitations of field testing, results in a need for emulated virtual deployments facilitating meaningful performance assessment and parameter tuning. Until recently, the only element painfully missing from the set was a detailed model of wireless channel (SMURPH was originally intended for modeling wired networks). Our most recent accomplishment is the addition to SMURPH a generic model of a radio channel, which allows us to create high-fidelity virtual networks consisting of hundreds or thousands of wireless devices. This greatly facilitates rapid application development, as well as hardware and protocol design.

2 SMURPH: the discrete charm of simulation

A simulation model in SMURPH consists of threads (simply called *processes*). Those threads execute in virtual time advanced in a discrete manner—upon the occurrence of events perceived by the model.

2.1 Threads

Fig. 1 shows a sample thread (process) in SMURPH. Its complete specification consists of the data part (the attributes declared within the `process` class) and the code (the argument-less method beginning with `perform`). The `states` declaration assigns symbolic names to the states of the FSM represented by the process. The first state on the list is also the initial state (the one in which the process finds itself immediately after creation). The activities of a process consist in responding to events, which can be awaited with `wait` and whose occurrence moves the process (transits the FSM) to a specific state (appearing as the second argument of the respective wait request). Note that `wait` can be a method of many different objects. All such objects belong to the same high-level class called an *activity interpreter* (`AI`). The collective pool of `AI`s emulates the world in which things happen. Another way of expressing this is to say that the activity interpreters are responsible for the flow (advancement) of (virtual) time.

The execution cycle of a process consists in being awakened in a specific state (which always happens in response to one awaited event),[7] executing the sequence of statements at that state (which typically includes wait requests declaring future waking conditions), and eventually going to sleep. The latter happens when the process exhausts the list of statements associated with the current state (hits the state boundary) or, alternatively, executes `sleep` (as within the `if` statement in state `Receive`—see fig. 1).

Note that a process may be waiting for a number of different events, possibly coming from different `AI`s. Such multiple requests form an *alternative* of waking conditions: the process will be awakened by the earliest of the awaited events in the state indicated by the second argument of the corresponding wait request. Once a process has been awakened (always by exactly one event), the remaining wait requests are forgotten, i.e., in each state the process builds its (possibly compound) waking condition from scratch.

[6] See `http://www.olsonet.com`.

[7] One exception is when the process is created, i.e., runs for the first time. But even then, there is an actual formal event (arriving from the process itself) responsible for starting the process up.

```
process SATReceiver (MStation) {
  Port *ISat;
  Mailbox *SM;
  void setup (int dir) {
    ISat = S->ISat [dir];
    SM = &(S->SyncMailbox);
  };
  states {WaitSAT, Receive};
  perform;
};
...
SATReceiver::perform {
  state WaitSAT:
    ISat->wait (BOT, Receive);
    Timer->wait (SATTimeout, Receive);
    wait (SIGNAL, WaitSAT);
  state Receive:
    SM->put (S->SatFlag);
    if (S->SatFlag) {
      wait (SIGNAL, Receive);
      sleep;
    }
    S->SatFlag = YES;
    proceed WaitSAT;
}
```

Fig. 1. A sample thread in SMURPH.

Several IPC (inter-process communication) tools built into SMURPH use the standard event triggering mechanism of activity interpreters. For example, processes can exchange signals. For the purpose of this operation, a process itself is viewed as an `AI`. In particular, the last wait request in state `WaitSAT` is addressed to *this* process and will occur when the process receives a signal. Another IPC tool, capable of passing possibly structured messages among processes, is `Mailbox`. For example, `SM` in state `Receive` points to a mailbox, and the `put` operation deposits an item into it.

2.2 The time model

In terms of actual execution (meaning how the collection of multiple SMURPH processes is run on the computer), only one process can ever be active at a time. Although parallel simulation can be of value in some areas, e.g., see [23], we have never missed it in performance studies of communication systems. A typical study in this area involves finding multiple points of a performance curve, and it makes much better sense to run multiple separate experiments on different CPUs (thus achieving a perfect speed-up) than trying to parallelize every single experiment (and unavoidably losing on synchronization).

The threads themselves, however, perceive their execution environment as being massively parallel. This is because, in accordance with the principles of discrete event simulation, the virtual time only flows while processes wait for something to happen. Thus, formally speaking, an arbitrary number of SMURPH processes can be active within the same unit of virtual time without any impact on the timing of their execution, which solely depends on the abstract models implemented by activity interpreters.

From the viewpoint of its actual execution, a process can only run if the "previous" process has gone to sleep. This paradigm resembles coroutines, as originally introduced in [5], except that, in our case, the control transfer is implicit (when a process goes to sleep it doesn't usually know which process will be run next), and there are multiple entry points (states) at which a coroutine (process) can be entered. This simple scheme circumvents all thread synchronization problems (and generally simplifies programming [19]) while not restricting at all the expressive power of the simulation model.

3 SIDE: virtual becoming real

If the SMURPH paradigm is to be used for writing real-life applications, the problem of non-preemptibility of threads must be taken into consideration. This problem is completely irrelevant in a simulated environment (because even a huge amount of real time spent by a thread on executing its state may result is no advancement of virtual time), but when virtual and real are merged into one, the execution time of threads will be perceived directly and, for example, may violate the real-time status of the controlled system.

3.1 Reactive systems

SIDE is intended for programming applications driving practical reactive systems [1, 22]. A reactive system may be roughly defined (at least for the purpose of this paper) as a collection of sensors and actuators. These two objects are very similar: their state is characterized by a single value. For a sensor, this value is modified by the environment and reflects the measured entity; for an actuator, the value is modified by the program and describes the physical action to be carried out by the sensor.[8]

Some reactive systems may actually pose hard real-time constraints. It may in fact happen that SMURPH threads are not adequate to fulfill such constraints, unless they are carefully programmed with those constraints in mind. This is because of their non-preemptibility. Formally, the scheduleability of threads (and thus the maximum response time of the program) is determined by the worst-case execution time of a thread state. Strictly speaking, this is not a property that goes well with *true* real-time applications.

Fortunately, most systems that are advertised are real-time are not. There is an overwhelming tendency to confuse “embedded” with “real-time,” perhaps because “real-time” sounds more dramatic and important. We agree with [2] that few reactive applications pose non-trivial real-time requirements, and most of them are in fact able to put up with remarkable sloppiness and non-determinism in response time. One anecdotal study, in which one of us participated as a hired consultant, involved the design of a TV settop box, with the project being announced as a real-time system development endeavor. Upon a closer scrutiny, it turned out that the single “real-time” requirement of the target system was its capability to respond to clicks on the remote.

On the other hand, note that SMURPH threads are easily malleable in that the granularity of their states is one of the design issues that can be addressed in many ways by the cognizant software developer. In particular, if the amount of computations in a given state appears too large, the state can be easily split into multiple states. There are also ways to relinquish control at short intervals—to let other threads run in the meantime.

3.2 The extent of changes

The complete transformation from SMURPH to SIDE essentially involved two steps:

1. turning virtual time into real time
2. providing a generic `AI` interface to real objects (external devices)

and has been accomplished with remarkably little effort. The first part was to make sure that when a wait request is issued to a timer (see the second statement in state `WaitSAT` in fig. 1), the delay is in fact real, i.e., the event is postponed until the real-time clock reaches a certain value. The second modification involved providing a special variant of the standard class `Mailbox`, which could be mapped to a peripheral device or a network socket. This solution was compatible with the standard behavior of mailboxes in SMURPH. The essence of the SIDE extension was to make it possible for the outside world to deposit data in a mailbox or to extract data from it. In particular, sensors (and also actuators) became expressible as mailboxes defined as shown in fig. 2.

The standard mechanism (already present in SMURPH) was used to trigger sensor events (when the value of sensor had changed). This is accomplished by executing `put` in method `setValue`, which deposits a dummy item into the mailbox thus triggering a `NEWITEM` event (see fig. 3). By making both operations (`getValue` and `setValue`) present in the same class, we avoid differentiating between a sensor and an actuator. Thus, for an actuator, `setValue` is executed by the application and the mailbox event is perceived by the mailbox interface to the world, while for an actuator, it is the other way around. The role of `NetAddress` and `mapNet` is to provide for a uniform way of assigning abstract sensor/actuator objects to their physical counterparts, and for consistently referencing them in the SIDE program. Fig. 3 shows a sample process handling two sensors and one actuator extracted from a program balancing the water level in a system of interconnected tanks.

The sensor abstraction is only one possible view of SIDE interface to the outside world. One powerful feature of the system is the option to map a mailbox to a network socket, which makes it possible to

[8] Note that, if needed, a complex sensor/actuator can be represented by a collection of simple ones.

```
mailbox Sensor {
  private:
    NetAddress Reference;
    int Value;
    void mapNet ();
  public:
    void setValue (int);
    int getValue ();
    void setup (NetAddress&);
};
...
void Sensor::setValue (int v) {
  Value = v;
  put (void);
};
int Sensor::getValue () {
  return Value;
};
```

Fig. 2. Sensor/actuator defined as a mailbox.

```
process PumpDriver (Pump) {
  Sensor *LLI, *RLI;
  Sensor M; /* an actuator */
  void setup () {
    LLI = S->LeftLevelIndicator;
    RLI = S->RightLevelIndicator;
    M = S->Motor;
  };
  states {WaitStatusChange, StatusChange};
  perform;
};
...
PumpDriver::perform {
  state WaitStatusChange:
    LLI->wait (NEWITEM, StatusChange);
    RLI->wait (NEWITEM, StatusChange);
  state StatusChange:
    if (LLI->getValue () < RLI->getValue ())
      M->setValue (PUMP_LEFT);
    else if (LLI->getValue () > RLI->getValue ())
      M->setValue (PUMP_RIGHT);
    else
      M->setValue (OFF);
    proceed WaitStatusChange;
};
```

Fig. 3. A sample thread in SIDE.

build distributed configurations of SIDE programs, possibly cooperating with non-SIDE clients. For example, SIDE was used in a Linux-based embedded system to implement a custom web server remotely supervising a collection of telecommunication devices.

4 PicOS: all in one piece

The primary problem with implementing non-trivial, structured, multitasking software on microcontrollers with limited RAM is minimizing the amount of fixed memory resources that must be allocated to a thread. One example of such a resource is the stack space, which must be preallocated (at least in some minimum safe amount) to every thread upon its creation. Note that the stack space is practically wasted from the viewpoint of the application: it is merely a “working area” needed to build the high-level structure of the program, and it steals the scarce resource from where it is “truly” needed, i.e., for building the “proper” data structures.

One popular system for programming small devices is TinyOS [18], where the issue of limited stack space has been addressed in a radical manner—by *de facto* eliminating multithreading. Essentially, TinyOS defines two types of activities: event handlers (corresponding to interrupt service routines and timer callbacks) and *tasks*, which are simply chunks of code that cannot be preempted by (and thus cannot dynamically coexist with) other tasks. Although programming within these confines need not be extremely difficult, we believe that our paradigm offers a significantly better compromise.

4.1 Stack-less threads

Owing to the fact that SMURPH processes can only be preempted at well-known points, i.e., state boundaries, they can effectively share a single stack. This is because a preempted process effectively returns (as a function), and its subsequent resumption consists in calling the function again (albeit at a possibly different entry point). The downside of this approach is that the automatic variables (i.e., ones allocated on the stack) receive a somewhat exotic semantics: their contents do not survive state transitions. Note that this does not mean that such variables are useless, only that they should be used in agreement with their (new) semantics.

```
process (sniffer, sess_t)
    entry (RC_TRY)
        data->packet = tcv_rnp (RC_TRY, efd);
        data->length = tcv_left (packet);
    entry (RC_PASS)
        if (data->user != US_READY) {
            wait (&data->user, RC_PASS);
            delay (1000, RC_LOCKED);
            release;
        }
        ...
    entry (RC_LOCKED)
        ...
    entry (RC_ENP)
        tcv_endp (data->packet);
        trigger (&data->packet);
        proceed (RC_TRY);
endprocess
```

Fig. 4. A sample thread in PicOS.

Threads in a PicOS application closely resemble SMURPH/SIDE processes (see fig. 4). The purely syntactical differences result from the economy of programming resources (C is used instead of C++ to avoid overtaxing the tiny RAM with the requisite run-time objects).

4.2 System calls

In a real operating system, a thread may become blocked not only because it decides to go to sleep (like exhausting the list of statement at its current state) but also when it executes a system call that cannot complete immediately. To make this work with PicOS threads, which can only be blocked at state boundaries, potentially blocking system calls must incorporate a mechanism similar to activity interpreters in SMURPH or SIDE. For illustration, see the first statement in state (entry) `RC_TRY`. Function `tcv_rnp` is called to receive a packet from a network session represented by the descriptor `efd`. It may return immediately (if a packet is already available in a buffer), or block (if the packet is yet to arrive). In the latter case, the system call will block the process (effectively forcing its function to return) with the intention of resuming it at the nearest moment when it will make sense to re-execute `tcv_rnp` (i.e., upon a packet reception). Then the process will be resumed in state `RC_TRY`, where it will invoke `tcv_rnp` again. The net outcome of this scenario is an implicit wait request to a conceptual `AI` handling packet reception, followed by `release`, which has the same effect as `sleep` in SMURPH/SIDE, i.e., it completes the execution of the current state (exits the process function).

Essentially, there are two categories of system calls in PicOS that may involve blocking. The first one, like `tcv_rnp`, may get into a situation when something needed by the program is not immediately available. Then, the event waking up the process will indicate a new acquisition opportunity (the failed operation has to be re-done). The second scenario involves a delayed action that must be internally completed by the system call before the process becomes runnable again. In such a case, the event indicates that the process may continue: it does not have to re-execute the system call. To keep the situation clear, the syntax of system call functions unambiguously determines which is the case. Namely, for the first type of calls, the state argument is first on the argument list, while it is last for the second type. Note that all system calls that can ever block take more than one argument.

4.3 Is PicOS real time?

PicOS being a close cousin of SIDE, we can repeat here the arguments given in sect. 3.1. If they appear unconvincing, please note that PicOS is in fact considerably more real-time than SIDE. This is because the interrupt service mechanism of PicOS is not part of the multithreading scheme, but operates independently of the thread scheduler. Consequently, the responsiveness of a PicOS application to external events (e.g., avoiding missing them) is completely independent from scheduleability of its threads.

In most practical cases; however, the system runs in a mode in which it can serve only one interrupt at a time. This is not a requirement for system consistency, but a practical restriction aimed at containing the stack size, which might otherwise exhibit a tendency to run away. On platforms with larger RAM, there is no formal reason why interrupts cannot arrive at the full speed of hardware. The non-preemptible model of PicOS multithreading can coexist with this feature without problems.

4.4 Footprint

So far, PicOS has been implemented on the MSP430 microcontroller family[9] and on eCOG1 from Cyan Technology.[10] Its primary role is to cater to tiny wireless applications, typically dealing with various sensors whose status is collected, possibly preprocessed, and forwarded to some sink nodes in a manner that may involve ad-hoc routing.

The size of the process control block (PCB) needed to describe a single PicOS process is adjustable by a configuration parameter, depending on the number of events E that a single process may want to await simultaneously. The standard setting of this number is 3, which is sufficient for all our present applications and protocols. The PCB size in bytes is equal to $8 + 4E$, which yields 20 bytes for $E = 3$.

The number of processes in the system (the degree of multiprogramming) has no impact on the required stack size, which is solely determined by the maximum configuration of nested function calls (plus one interrupt). As automatic variables are of limited usefulness to processes (sect. 4.1), the stack has no tendency to run away. 128 bytes of stack size is practically always sufficient. In many cases, this number can be reduced by half.

PicOS has an efficient memory allocator (`malloc/free`), which is mostly used by its layer-less networking module (dubbed TCV) to maintain pools of packet buffers. We have a full-size implementation of an ad-hoc routing protocol [21], along with a family of non-trivial applications taking the full advantage of multi-hop ad-hoc networking, operating on MSP430F148 with 2KB or RAM. It is conceivable to have an operable node with 1/2 of that memory.

5 Summary

The programming paradigm of SMURPH presented in this paper can be succinctly characterized as "coroutines with implicit control transfer and multiple entry points." Over several years, this paradigm has been recurring to us as a very convenient tool in those areas of computation that involve lots of events and little number crunching. We have found it useful for talking about such systems (i.e., specifying them), building their detailed simulation models, and (recently) implementing them—at least on a certain class of CPUs.

Notably, despite the tremendous advances in computing technology, which have brought about unbelievable cost reduction for the devices that not so long ago would be considered extremely advanced and prohibitively expensive, the trivially small microcontrollers do not want to disappear from the spectrum. On the contrary: their ever decreasing cost has enabled new applications and, recently, triggered unprecedented interest in large and inexpensive networks of sensors. It seems that PicOS provides a friendly environment for implementing efficient and effective solutions in this expanding niche.

Our present project aims at direct simulation (or rather emulation) of PicOS software in SMURPH. As a prerequisite, we have augmented SMURPH with a sophisticated meta-model of a radio channel. Now, owing to the obvious similarities between PicOS and SMURPH, the project practically boils down to creating a generic model of a radio device. Then, PicOS internal interfaces to radio modules will have to be reorganized, such that they can be automatically identified and redirected to the models.

[9] See `http://www.msp430.com/`.
[10] See `http://www.cyantechnology.com/`.

References

1. K. Altisen, F. Maraninchi, and D. Stauch. *Aspect-oriented programming for reactive systems: a proposal in the synchronous framework,* Research report no tr-2005-18, Verimag CNRS, November 2005.
2. A. Benveniste and G. Berry. *The synchronous approach to reactive and real-time systems,* Proceedings of IEEE, 79(9):1270–1282, 1991.
3. M. Berard, P. Gburzyński, and P. Rudnicki. *Developing MAC protocols with global observers,* In: Proceedings of Computer Networks'91, pages 261–270, June 1991.
4. D. R. Boggs, J. C. Mogul, and Kent C. A. *Measured capacity of an Ethernet: Myths and reality.* WRL research report 88/4, Digital Equipment Corporation, Western Research Laboratory, 100 Hamilton Avenua, Palo Alto, California, 1988.
5. O. J. Dahl and K. Nygaard. *Simula: A language for programming and description of discrete event systems,* Introduction and user's manual, 5th edition, Norwegian Computing Center, Oslo, 1967.
6. W. Dobosiewicz and P. Gburzyński. *Improving fairness in CSMA/CD networks,* In: Proceedings of IEEE SICON'89, Singapore, July 1989.
7. W. Dobosiewicz and P. Gburzyński. *Performance of Piggyback Ethernet,* In Proceedings of IEEE IPCCC, pages 516–522, Scottsdale, AZ, March 1990.
8. W. Dobosiewicz and P. Gburzyński. *On the apparent unfairness of a capacity-1 protocol for very fast local area networks* In Proceedings of the Third IEE Conference on Telecommunications, Edinburgh, Scotland, March 1991.
9. W. Dobosiewicz and P. Gburzyński. *An alternative to FDDI: DPMA and the pretzel ring,* IEEE Transactions on Communications, **42:** 1076–1083, 1994.
10. W. Dobosiewicz and P. Gburzyński. *On two modified Ethernets,* Computer Networks and ISDN Systems, pages 1545–1564, 1995.
11. W. Dobosiewicz and P. Gburzynski. *Protocol design in SMURPH,* In J. Walrand and K. Bagchi, editors, *State of the art in Performance Modeling and Simulation*, pages 255–274. Gordon and Breach, 1997.
12. W. Dobosiewicz and P. Gburzyński. *The spiral ring, Computer Communications*, 20(6):449–461, 1997.
13. W. Dobosiewicz, P. Gburzyński, and V. Maciejewski. *A classification of fairness measures for local and metropolitan area networks,* Computer Communications, 15:295–304, 1992.
14. W. Dobosiewicz, P. Gburzyński, and P. Rudnicki. *On two collision protocols for high-speed bus LANs,* Computer Networks and ISDN Systems, **25**(11):1205–1225, June 1993.
15. P. Gburzyński. *Protocol design for local and metropolitan area networks,* Prentice-Hall, 1996.
16. P. Gburzyński and J. Maitan. *Simulation and control of reactive systems,* In: Proceedings of Winter Simulation Conference WSC'97, pages 413–420, Atlanta, Georgia, December 1997.
17. P. Gburzyński, J. Maitan, and L. Hillyer. *Virtual prototyping of reactive systems in SIDE,* In: Proceedings of the 5th European Concurrent Engineering Conference ECEC'98, pages 75–79, Erlangen-Nuremberg, Germany, April 1998.
18. J. Hui. *Tinyos network programming (version 1.0), 2004,* TinyOS 1.1.8 Documentation.
19. E. A. Lee. *The problem with threads,* IEEE Computer, **39**(5):33–42, 2006.
20. Polish-Japanese Institute of Information Technology. *Performance features of global states based application control,* In: Proceedings of Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, February 2006.
21. A. Rahman, W. Olesinski, and P. Gburzynski. *Controlled flooding in wireless ad-hoc networks,* In Proceedings of IWWAN'04, Oulu, Finland, June 2004.
22. B. Yartsev, G. Korneev, A. Shalyto, and V. Ktov. *Automata-based programming of the reactive multi-agent control systems,* In: International Conference on Integration of Knowledge Intensive Multi-Agent Systems, pages 449–453, Waltham, MA, April 2005.
23. L. Zhu, G. Chen, and B. Szymanski *et al.* *Parallel logic simulation of million-gate VLSI circuits,* In: Proceedings of MASCOTS05, pages 521–524, Atlanta, GA, 2005.