Specification Design Method for Reactive Embedded Systems: A Case Study

Benny Shimony [*] **Ioannis Nikolaidis** [*] **Pawel Gburzynski** [*]
Eleni Stroulia [*]

[*] *University of Alberta, Department of Computing Science, Edmonton, AB, CANADA T6G 2E8 (e-mail: {shimony,yannis,pawel,stroulia}@cs.ualberta.ca).*

Abstract: We report on a case study that evaluates a top-down design approach geared to the wireless sensor network (WSN) domain. Starting from a high-level specification tool based on the DisCo system, we evaluate an intermediate level of modeling based on an FSM automaton description using two types of tools and translation methods. The first is a simulation and coordination environment, and the second is a timed-automaton form with the UPPAAL tool. In both methods we incorporate model components that represent the environment, resulting in a more realistic operational estimate of the design by simulation and increasing our ability to analyze the feasibility of various design facets, i.e., real time behavior, energy use, computation, memory resources etc.

Keywords: Embedded systems, reactive software, software design, wireless networks.

1. INTRODUCTION

Wireless sensor networks (WSN) are instances of highly distributed, large scale, unreliable systems. Individual nodes are simple computing devices with severely limited resources. In addition, the wireless medium introduces complications, such as sensitivity to the physical conditions in the form of channel interference, network deployment density, reliance on non-deterministic medium access protocols, etc. Furthermore, the development cycle requires field testing and debugging that are labor-intensive, resulting in increased development time and cost, which makes it difficult to authoritatively test the system before its target deployment.

We describe a case study that evaluates a design methodology geared to the WSN domain. The strength of the proposed methodology is in its integrated design approach, i.e. facilitating a development process from specification down to the implementation level with tools supporting translation, testing and debugging. Moreover, we believe that the methodology is flexible and configurable, so that it can accommodate different levels of the modeling and design process. The method presented here consists of three phases. The first phase, a high–level specification of the system, is currently based on a tool named Distributed Cooperation (DisCo), see Kurki-Suonio (2003). DisCo assumes an action–oriented model of computation. Such models differ from explicit flow control models; in particular, the DisCo system allows for a top–down design approach in incremental steps of refinement and composition based on components or ‘layers’ of behaviors which replace the traditional procedural modules.

The final (third) phase for the intended implementation target is related to the capabilities of the PicOS operating system, see Akhmetshina et al. (2003), a locally developed FSM–oriented light–weight OS for WSNs. The intermediate level, which bridges the gap between the two, translates the specification to an automaton–based model. We show that, at the automaton level of modeling, the specification can be tightly correlated to the actual system environment, using an operational interpretation of the target environment and tool capabilities, such as model–checking and simulation. The environment component allows us to evaluate resource feasibility in various aspects of design: timing, energy efficiency, computation/memory, etc.

Among the advantages of this modeling method is its assumption that the system is *closed*, i.e., the designed (proper) system *as well as* the environment in which it will operate are modeled *together*. The main benfit of incorporating the environment into the specification is to promote realistic modeling. In the domain of reactive systems, such as those consisting of wireless sensors, one advantage brought by including the environment in the model is the possibility of expressing elaborate properties of the communication channel (e.g., interference, mobility), or capturing user interaction and operation patterns. Such elaborate models have already been recognized and incorporated in simulation environments for system performance evaluation, see Gburzynski and Nikolaidis (2006). We argue that they could be meaningfully incorporated at an early design stage as part of the specification.

In this paper we report on our evaluation of two methods of testing and transforming the high level specification into the intermediate level of modeling:

(1) a basic simulator/coordination environment based on the DisCo tool set, see Aaltonen et al. (2001), and
(2) a translation/reduction to a timed automaton model using the UPPAAL language and tool, see Behrmann et al. (2004).

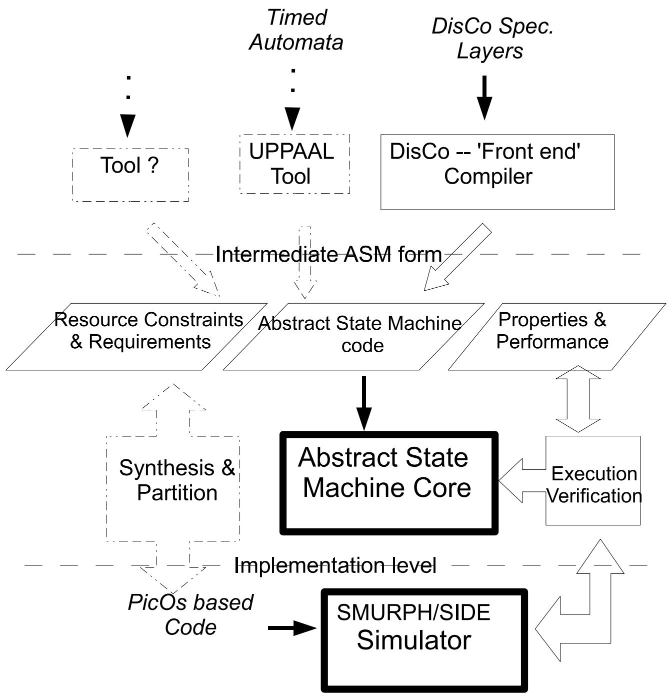


Fig. 1. Overview of the design methodology.

Ongoing work is targeting the extension of the presented basic method of translation towards its complete automation.

The rest of the paper is structured as follows. In section 2 we present a background of the design architecture, and in section 3 we show a basic design in the DisCo system, which is used as a case study. In section 4 we detail the domain specific Simulation and Coordination Environment and describe its operational interpretation model and algorithmics. Section 5 discusses the UPPAAL example design focusing on the basic translation methods to the automaton form. In section 6, we describe the results of feasibility tests applied to the example models in the two translation methods. Section 7 reviews the related work on design tools and methods (for embedded systems, including real time), and in section 8 we conclude the paper with a brief discussion of future work.

2. BACKGROUND

An overview of our proposed methodology is depicted in Fig. 1. We have selected the automaton–based transformation as the basis for the modeling method and as an intermediate level of modeling. First, it is a sound and proved modeling method being conceptually easy to grasp. Second, it lends itself to a wide range of specification methods/paradigms of design that can be translated into this format. Moreover, within its framework, we can utilize the vast options of modeling tools, e.g., UPPAAL, see Behrmann et al. (2004) (our choice in this paper), or ASM languages, see Farahbod et al. (2007); Ouimet and Lundqvist (2008), and take advantage of its capabilities for testing and verification. Lastly, this approach is already close to the FSM paradigm of PicOS, which constitutes our implementation target.

We hope that all these considerations will facilitate a separation of concerns in the software and embedded design. We intend to extend the evaluation of high-level specification methods over other methodologies. With the intermediate level model we can integrate the design with other tools and verify the functional properties of that design. At this stage we can introduce additional non-functional properties, such as resource constraints, timing considerations and energy aspects of the design (as described in the example presented in this paper). Moreover, the intermediate level model can be assigned network and communication abstractions which can vary by the detail level of the specifications.

In the final stage, the intermediate automaton models need to be translated to a 'de-centralized' architecture, i.e., to the target distributed system. The final result will produce multiple automaton models representing the deployed nodes and the environment. The PicOS threads are closely related to an FSM format, and the special semantics of PicOS threads and specific embedded system services need to be accounted for on each of the deployed nodes. To support testing of the designs, we plan using the PicOS simulation counterpart: SMURPH/VUEE, see Gburzynski and Nikolaidis (2006), as a feedback to the defined properties of the model. Thus, for example, in the emulation setting the environment models can be translated to SMURPH 'Client' models to attain realistic simulations, and dynamic properties can be verified using SMUPRH 'Observer' models.

Next, we present the key features of our choices for specification, refinement, and implementation.

2.1 DisCo

The high-level specification method is based on a specification tool named Distributed Cooperation (DisCo), see Kurki-Suonio (2003). The DisCo language is built on two basic constructs: 'Actions' and typed 'Objects'. Actions (also termed 'Joint Actions') have an operational interpretation: they are intended to represent the operations for the system, as it changes its state from one to another. Objects are containers of data specified by class types. The system state is the combination of all class instances along with the values of internal variables. The model of execution is assumed to involve *interleaved atomic* actions. System executions (or behaviors) are sequences of actions and states interleaved in some order, whereby an action is guaranteed to be completed without an interference. At each step, an action can be preformed if its *guard* condition is enabled and all of its participating objects are available.

DisCo follows the formal semantics of Temporal Logic of Actions (TLA), see Lamport (1994), and ensures that refinement and composition of component-'layers' preserves basic 'safety' properties that have already been tested. This helps the correct-by-design approach, i.e., integration with other components does not fault the system.

DisCo allows for the incorporation of an 'Environment' model, which can include, for example, components to capture discrete user interaction and operation patterns, or to employ external functions to capture continuous-time physical phenomena, like wireless channel behavior. The

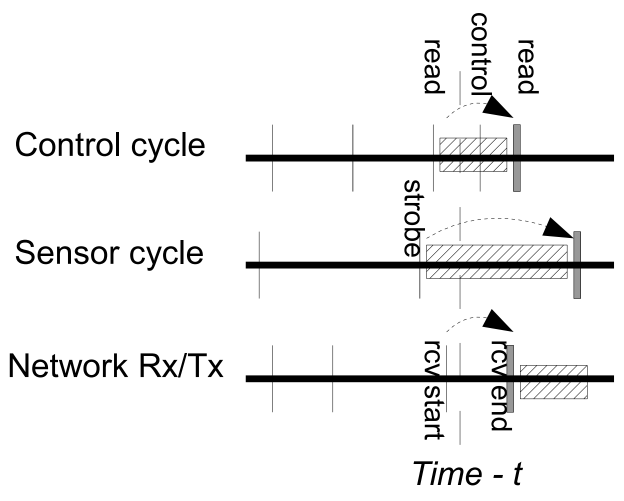


Fig. 2. ToyBot timing requirements.

combined support of continuous and discrete time models is usually termed *hybrid* modeling.

We note that DisCo specifications enable the definitions of deadline relations between actions that share a 'time' variable. Basically, there are two types of deadline relations: type-I: *not later than* (event time $\leq$ Deadline Time) and type-II: *not earlier than* (event time $\geq$ Deadline Time). Periodic events are combinations of the two types and are reset each time the event occurs. Fig. 2 shows an example of timed execution defined for our ToyBot. Note that the top two relations are type-I, while the third one is type-II.

2.2 UPPAAL

UPPAAL, see Behrmann et al. (2004), is a language and verification tool for real-time systems, which is based on the theory of Timed Automata. A timed automaton is a finite-state machine augmented with clock variables. The UPPAAL language offers additional features, such as bounded integer variables, which can be used, as in a programming language, to update the system state. Another special flavor of the UPPAAL system is that time variables assume discrete values, instead of real values in the general model. In addition the UPPAAL tool includes a query language to specify properties that can be tested by the model-checking capabilities.

A system is modeled as a network of several automata working in parallel. The automata can change their state by 'firing' edges (or transitions) that change their locations and set new values to variables. A fundamental characteristic of the network is that clock variables progress at the same rate. The system state is defined by the locations of all automata, the values of the discrete variables, and the clock constraints values. The automata may synchronize through special 'channel' variables. In effect, the synchronization actions produce a joint state change among the synchronized automata. This last point is a major semantic similarity to the 'Joint Action' principle of the DisCo system. A transition (either synchronous or not) is enabled by a predicate 'guard' condition. Additional features make it possible to define priorities and scheduling properties. Committed locations model atomic actions, which are acted upon immediately (before any other non-Committed actions). In addition, three urgency mechanisms: location invariants, location urgency, and urgent channels cater to special scheduling requirements.

2.3 PicOS

The special characteristics of WSN have driven efforts to simplify application development. The most popular WSN OS platform, TinyOS, see Levis et al. (2005); Hui (2004), provides components for self sustained execution environments, which include abstractions for communication, scheduling and embedded services. TinyOS programs are based on single-threaded code with multiple-event handlers. Event handling components need to be glued together through interfaces in a style reminiscent of coordination models and languages. This is in contrast to the more familiar model of multi-threading and blocking system calls, which are usually deemed as being resource expensive. An answer to these concerns is PicOS, see Akhmetshina et al. (2003), a locally developed, low-overhead sensor operating system. Similar to TinyOS, PicOS is also centered around events, but it provides a lean mechanism of multi-tasking featuring a finite state machine (FSM) type of abstraction.

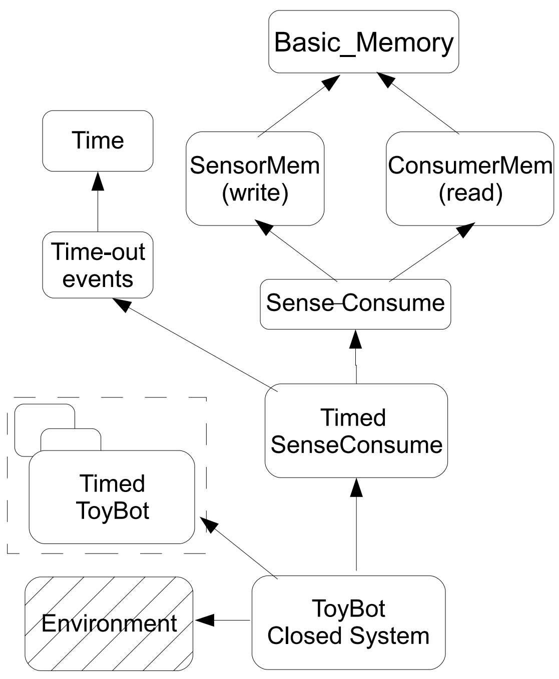
While PicOS solves many of the challanges of efficient programming for the individual nodes, it lacks support for a system level view of the whole solution at the distributed (network) level. Hence the need to assist the development with a global-view methodology as the one presented in this paper.

The execution dynamics of PicOS resemble a co-routine flow of control, which is hard to track and non-deterministic. Moreover, efficient component re-use requires expertise and a need to understand the detailed architecture of the system. Our present work attempts to address these issues by providing a model based development environment for WSN. The environment should be configurable, based on sound formalisms, and close to the semantics. We hope to facilitate modularity and easy code reuse through design methods oriented towards components and aspects, and provide tools for automatic synthesis of code, where the target object programs are PicOS applications (so-called praxes, see Boers et al. (2009)). Moreover, we hope to support implementation level testing through simulations coupled to the property checking capabilities at the model level.

3. SPECIFICATION: AN EXAMPLE

Consider a 'ToyBot' (robot) node with sensing and reporting operations and a movement component on a 2D grid. The example is inspired by the model in Kurki-Suonio (2003), which we refine to illustrate the introduction of the environment and the resource consumption into the model. The layered structure of the specification is depicted in Fig. 3.

The *Sense–Consume* layers specify the acquisition of data from a sensing operation, buffering data in local memory and the consumption of data by some 'abstract' consumer. The *Memory* buffer acts as a mediator between the two actions. The code snippet (the right part of Fig. 3) demonstrates the syntax of layering. The *Basic_Memory* layer



```
layer Basic_Memory is
 class Memory is
   cap : integer := 4096;
   ptr : integer := 0;
 end;
 action ReadMemory (m:Memory; len: integer) is
   when (m.ptr >= len) do
    m.ptr := m.ptr - len;
   end;
 action WriteMemory (m:Memory; len: integer) is
   when (m.ptr < m.cap - len) do
    m.ptr := m.ptr + len;
   end;
```

```
layer ConsumerMem is import Basic_Memory;

 class ConsumeState is
  status : (ready, busy);
 end;

 refined ConsumeMemRdy(m, len, c:ConsumeStatus) of
   ReadMemory(m,len) is
   when ... (c.status'busy)
   do ...
    c.status->ready();
   end;
```

Fig. 3. The *Sense–Consume* layers (left) and snippet of memory model refinement (right).

defines the *ReadMemory* and *WriteMemory* operations (at this level we model only the capacity and utilization of the memory). The *ConsumerMem* layer extends and refines the memory-read operation (note that it imports the *Basic_ Memory* layer) by a two phased action. For brevity, we detail only the second phase in action *ConsumeMemRdy*. The '...' syntax indicates the inclusion of all conditions and operations of previous layers. The *SensorMem* layer (not shown here) refines the memory-write operation. Finally, both *ConsumerMem* and *SensorMem* layers are merged in the next layer, *Sense-Consume*. The *TimedSenseConsume* layer imports a generic real-time event, which is then 'super-imposed' or 'associated' with the *SensorMem* and *ConsumeMem* operations (for periodic data strobes and delayed two phase data consumption). The *TimedToyBot* layers (the dashed box on the left) define the node's mobility operations, similar to the example in Kurki-Suonio (2003), incorporating a control loop: a read action to get movement data, and computing the control action that sets the engine power.

The *Environment* component includes a model to react to the oeration of all nodes and to simulate an environment to all its inputs. Thus, for example, the environment model translates the actuator values of engine power to acceleration/speed/location through a simple physical simulation. The environment component is unique in that it accommodates external modules which can be invoked to participate in the simulated environments. In the specification syntax these modules need to be defined by 'function' constructs.

4. OPERATIONAL INTERPRETATION

The DisCo joint-action abstraction represents a system where executions are interleaved sequences of atomic *actions*. It was shown in Kurki-Suonio (1993) that using an interleaved model does not constraint the implementation, or (in our case) simulation, from modeling concurrent execution. Namely, unrelated actions and object participants are allowed to be operating at the same time. Moreover, in constructing the model, the granularity of actions can be freely defined. Hence, if the start and end of an event is defined by two separate actions, then several such events can occur concurrently. To capture the timing requirements included in the models, the simulation uses a non-decreasing global time variable, and each action of the DisCo model implicitly includes a time variable expressing its activation time. The basic operational interpretation of the joint-action abstraction assumes that actions are, in essence, provide a synchronization mechanism between their participating objects. In this interpretation, the simulation requires that each of the model objects (e.g., memory, sensors) be associated with a resource, like a processor or an independent hardware sub-module. Thus, in the example model, the consumer objectc in the *ConsumerMem* read action (Fig. 3) can belong to a different processor than that of the Sensor object of the *SensorMem* action, so as to simulate communication by means of their joint action.

4.1 Simulation and Coordination Environment

The simulation and coordination environment utilizes the principles of the scenario animator tool of the DisCo toolset, see Aaltonen et al. (2001), to execute simulation runs. In this format, the generated Java classes (from the tool-compiler) are the basic executed components, and they are augmented with 'foreign-coded' functions that can model the environment.

The simulation algorithm emulates the interleaving model. At each step one of the enabled action is non-deterministically selected with its participating objects. An action is enabled only if there are available objects to participate in it. At each time slot, several actions can be performed, given that they are not dependent on each others objects' processor resources. Once there are no other possible actions in a time slot, the simulator performs an 'Environment' action as part of the closed world assumption.

5. DISCO TRANSLATION TO UPPAAL AUTOMATON

First we shall give a brief motivation for our translation method. As noted, the basic semantics of the joint-action in DisCo are similar to the synchronized operation of automata: both preform simultaneous updates to several objects (or automata) in a single 'atomic' step. While in the UPPAAL model the automata include both the state of objects (such as local data) and actions (such as transition edges), the DisCo model defines separately objects and actions. Objects are data containers, while actions are parameterized with participating objects. Thus, in the translation method, we need to aggregate for each object-based automaton the set of joint-actions that it participates in. Other aspects of the action translation method concern DisCo's joint-action guard section which can be directly translated to the automaton's edge guard conditions. The body of joint actions performs updates to the state/data of the participating objects. In the UPPAAL language, the basic method of sharing data between automata is through global variables. For this purpose, any data passed between objects (i.e., non-local) needs to be designated as a global variable, which is accessible to other automata. In essence these, can be seen as a type of basic messaging among objects. The general method of translation needs to handle some additional incompatibilities and provide for a more formal description of the reduction method. These issues are beyond the scope of the present paper. We demonstrate an example translation to a part of the model presented in section 3. The snippet in Fig. 4 presents the action headers that correspond to the *TimedSenseConsume* layer of Fig. 3.

Notice that the *WriteMemory(..)* operation now includes the **s:Sensor** object, and the consumer read operations: *ConsumeData(..)* and *ConsumeRdyFromMem(..)* include the **c:ConsumeState** object. The consumer operation is split into two: *ConsumeData* starts a time period (specified by the **dlTo** parameter), *ConsumeRdyFromMem* is activated after a time-out expires. Note that only *ConsumeRdyFromMem* directly refines the *ReadMemory* operation of the higher level layers. The automaton network representing the *TimedSenseConsume* layer is depicted in Fig. 5.

The *BasicMemory* automaton has two synchronized transition edges: the read operation on the right and the write operation on the left. The read operation is synchronized with the *ConsumerMem* automaton through the channel variable **rd_trig** tagged with '?' indicating that it is the receiver of the signal. The write operation is synchronized with the *SensorMem* automaton through the channel variable **wr_trig**, similarly tagged with '?' to mark it as the receiver. The transmitting channel sources are tagged with '!' in the *ConsumerMem* and *SensorMem* automata, which means that they are the transmitters of the signal. Also

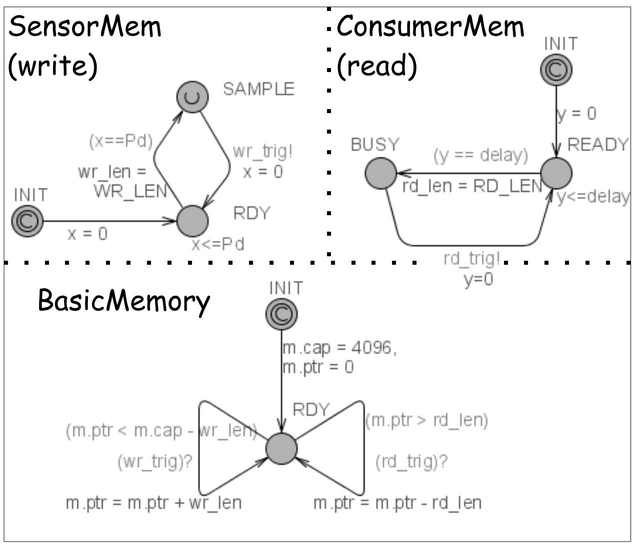


Fig. 5. *TimedSenseConsume* UPPAAL automaton

note that the variable **m** is a local variable of the *BasicMemory* automaton and it directly corresponds to the *Memory* class data of layer *BasicMemory* in Fig. 3.

Next, we consider the operation of each automaton in more detail. The edges of *SensorMem* correspond to the *WriteMemory(..)* operation. The variable **x** corresponds to the local clock and the transition from the INIT state resets its value. The edge from state RDY to SAMPLE is activated once the guard condition **(x==Pd)** holds. **Pd** is a constant defined as the duration of the sampling period. In addition, the condition **(x<=Pd)**, defined by the RDY location, is an invariant. That invariant implies that the transition from RDY to SAMPLE must be fired once **x** equals the sampling period (and thus no deadlock is possible). The edge transition updates the global variable **wr_len**, which sets the write block length. The value is later read by the *BasicMemory* automaton when the synchronized operation is activated. The 'U' mark in state SAMPLE signifies that the state is *urgent*, i.e., time should not pass in this state. Thus, the next transition from state SAMPLE to RDY occurs in the same 'virtual' time as the previous step (although in an ordered fashion) and triggers the synchronized operation with the signal **wr_trig!**, which fires the left edge of *BasicMemory* as well.

In the *ConsumerMem* automaton, the variable **y** corresponds to the local clock, and the transition from the INIT state resets its value. The automaton also includes the states BUSY and READY with the edges between them corresponding to the actions *ConsumeData(..)* and *ConsumeRdyFromMem(..)*. The edge from READY to BUSY is activated only after a time-out clock condition *(y==delay)* is satisfied (the guard condition of the edge). The edge transition also updates the global variable **rd_len** to set the read block length. The value is later read by the *BasicMemory* automaton, once the edge from BUSY to READY activates the synchronized operation (with signal **rd_trig!**) and the *BasicMemory* automaton fires the right edge action.

One additional detail to note in the above example is that the BUSY to READY edge in the *ConsumerMem*

```
layer TimedSenseConsume is
 refined ConsumeData( dlTo:real; c:ConsumeState ) of
   ConsumeData(c)
 refined ConsumeRdyFromMem( m:Memory; len:integer; c:ConsumeState ) of
   ConsumeMemRdy&ReadMemory (c, m, len)
 refined WriteMemory ( m:Memory; len: integer; s: Sensor) of
   WriteMemory ( m, len )
```

Fig. 4. Snippet of *TimedSenseConsume* Layer Actions.

automaton is *urgent*, which means that it should be 'fired' as soon as its guard is enabled. In our case, the guard is a combination of the synchronized action edges and includes the guard condition of a *BasicMemory* edge, i.e., **(m.ptr > rd_len)**. The above example also shows the possibilities of timing translation, periodic timing in the case of *SensorMem* and no earlier deadline, using a combination of clocks and urgent edges in the *ConsumerMem* automaton.

6. SIMULATION AND EARLY FEASIBILITY TEST RESULTS

We applied the defined timing scheme of the example to a 2-level memory hierarchy model (RAM and Flash), with the objective to determine the buffering needs. Namely, the model assumed that the non-volatile Flash memory is the consumer of data and, hence, it refines the *ConsumerMem* layer introducing an external module to model the write timing behavior of the Flash. The test closely emulates the real-time constraints from Gburzynski and Kaminska (2008).

The results in Table 1 compare the two translation schemes introduced in previous sections: the simulation runs executed with the accommodated DisCo tool-set environment, and the UPPAAL model checker tool.

The consumer read operation is a two stage operation (Read and Rdy) which is added in the timing module of the Flash device. The scenario depicts a simple case of periodic reads and writes: sampling 12 bytes every 8 time units, and reading a 40-byte block (to be subsequently written to the Flash) every 20 time units. The base case validates the integrity of the two models, showing the same result for this deterministic scenario (which requires a maximum buffer size of 54 bytes). The second row llustrates the variance in a scenario with random consumer timing, which was modeled using the simulation tool. The delay is a random function with an average of 20 time units and the variance of 7; the maximum buffer size observed in this case was 84 bytes. The third row corresponds to the case where an offset between the consumer read clock and the sensor write clock was added in the UPPAAL model. As shown, this can increase the amount of buffer space required. The fourth and subsequent rows show the results of a jitter and delay model applied in the UPPAAL tool.

A point to notice is that the UPPAAL tool checks the entire state space; hence, its results are the definitive property of the system, whereas the simulation runs may only amount to several random (accidental) scenarios which not necessarily explore all extreme cases. On the other hand, the UPPAAL models can not use random models.

7. RELATED WORK

The AADL (Architecture Analysis and Design Language), see SAE International (2006), is an industry supported standard to model and analyze the design of embedded systems. The standard also suggests a top-down design approach with the intention to model the whole system and to guarantee its properties. The standard includes tools and extensions to support analysis, real-time design, distributed middle-ware, and automatic code generation from the early stages of design. Yet, the language is primarily concerned with the non-functional properties of components and their interfaces where the behavioral properties need to be given by source code in another programming language. Moreover, AADL applies to large embedded systems with adequate resources, and has not beenapplied to the WSN domain.

The work by the ISIS-group at Vanderbilt University, see Porter et al. (2009), provides an extensive architecture for integrated development environment of embedded control systems. The system includes a suite of design tools and languages, and it partially supports AADL standards. The ISIS tool chain aims to provide a unified environment that can integrate disparate aspects, e.g., safety-critical embedded designs, while providing separation of concerns between software and engineering, e.g., hardware and platforms. The graphical architecture description language ESMoL incorporates designs from the Simulink and Stateflow sub-languages which are the basis of the modeling environment. In addition, the ISIS Generic Modeling Environment tool (GME) can also be included for high level models, see ISIS (2008), together with other domain specific modeling languages. The tools support a wide range of hardware and platforms and various communication types, buses, Ethernet, etc. Currently, real time control design is based on time triggered architectures, see Kopetz and Bauer (2001). Our proposed architecture shares many of the objectives and concepts of the ISIS group project, while intentionally specializing for WSN applications (deployment in simple motes with limited resources communicating over a wireless channel). Also, the present version of ESMoL does not use formal requirements as the basis of design as ours does.

Other notable approaches in the WSN domain include macro-programming languages and environments, their goal being to enable programming a whole solution at the network level. Those approaches apply mostly to classes of related/similar problems in WSNs. For example, TinyDB, see Madden et al. (2005), is intended for hierarchical sensor data collection through the use of SQL language and a Database abstraction. The solutions provided (through compilation) are a family of similar protocols based on

Table 1. Experimental Results

Evaluation Type (12 byte samples, 40 byte reads)	DisCo Sim-Coord Buffer Size	UPPAAL Model Buffer Size
Base Case	read/write periods: 8/20	read/write periods: 8/20
	52	52
Random Case	read/write periods: 8/20±7	read/write periods: 8/20
	84	–
Offset ≤ 40	read/write periods	read/write periods
	–	60
Offset ≤ 40 **& Jitter** ≤ 8	read/write periods	read/write periods: 8/12-20
	–	84
Offset ≤ 40 **& Jitter** ≤ 10	read/write periods	read/write periods: 8/10-20
	–	84
Offset ≤ 40 **& Jitter** ≤ 12	read/write periods	read/write periods: 8/8-20
	–	84

TinyOs implementations that all fall under the umbrella of data aggregation techniques and are based on the formation of spanning trees across the network.

8. DISCUSSION AND FUTURE DIRECTIONS

The development of reactive embedded systems is particularly sensitive to real time and resource requirement considerations. By bringing them into the model at the early stages of specification, we are able to test for the feasibility of the design before committing to detailed implementation. The demonstrated results are very preliminary. We plan to extend the modeling capabilities of the hybrid environment by including additional models such as those of the wireless channel, user interaction properties, energy consumption, and energy management components. One of the major challenges is to extend the specification methods to a distributed setting, i.e., a network of nodes, and to construct new systems/applications and control solutions. Moreover, we are interested in integrating existing library components into the models. These mostly include network algorithms and abstractions which have proved implementations. Three notable examples are: routing protocols, see Gburzynski et al. (2007), self-stabilizing robust network protocols (e.g., for leader election), localized network abstractions (e.g., 'neighborhood' clustering abstractions, see Whitehouse et al. (2004)). Finally, the design of automated compilation tools is an integral part of the environment. As a first step, we intend to complete the DisCo specification translation into UPPAAL format. At a later stage, the compilation into PicOS-based implementations is our goal.

REFERENCES

Aaltonen, T., Katara, M., and Pitkänen, R. (2001). DisCo toolset–the new generation. *Journal of Universal Computer Science*, 7(1), 3–18.

Akhmetshina, E., Gburzynski, P., and Vizeacoumar, F. (2003). PicOS: A tiny operating system for extremely small embedded platforms. In *Proceedings of ESA 2003*, 116–122. Las Vegas.

Behrmann, G., David, R., and Larsen, K.G. (2004). A tutorial on UPPAAL. 200–236. Springer.

Boers, N., Gburzynski, P., Nikolaidis, I., and Olesinski, W. (2009). Developing wireless sensor network applications in a virtual environment. *Telecomm. Syst.* To appear.

Farahbod, R., Gervasi, V., and Glässer, U. (2007). CoreASM: An extensible ASM execution engine. *Fundam. Inf.*, 77(1-2), 71–103.

Gburzynski, P. and Kaminska, B. (2008). Testing real-time properties of embedded systems. In *Proceedings of ESA 2008*. Las Vegas.

Gburzynski, P., Kaminska, B., and Olesinski, W. (2007). A tiny and efficient wireless ad-hoc protocol for low-cost sensor networks. In *Proceedings of DATE 2007*, 1562–1567. Nice, France.

Gburzynski, P. and Nikolaidis, I. (2006). Wireless network simulation extensions in SIDE/SMURPH. In *Proceedings of WSC 2006*, 2225–2233.

Hui, J. (2004). TinyOS network programming (version 1.0). TinyOS 1.1.8 Documentation.

ISIS (2008). GME: Generic modeling environment. `http://repo.isis.vanderbilt.edu`

Kopetz, H. and Bauer, G. (2001). The time-triggered architecture. *IEEE, Special Issue on Modeling and Design of Embedded Software.*

Kurki-Suonio, R. (1993). Stepwise design of real-time systems. *IEEE Transactions on Software Engineering*, 19(1), 56–69.

Kurki-Suonio, R. (2003). Action systems in incremental and aspect-oriented modeling. *Distrib. Comput.*, 16(2-3), 201–217.

Lamport, L. (1994). The temporal logic of actions.

Levis, P. et al. (2005). TinyOS: An operating system for sensor networks. In W. Weber, J. Rabaey, and E. Aarts (eds.), *Ambient Intelligence*, 115–148. Springer.

Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2005). TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1), 122–173.

Ouimet, M. and Lundqvist, K. (2008). The timed abstract state machine language: Abstract state machines for real-time system engineering. *Journal of Universal Computer Science*, 14(12), 2007–2033.

Porter, J., Karsai, G., Völgyesi, P., Nine, H., Humke, P., Hemingway, G., Thibodeaux, R., and Sztipanovits, J. (2009). Towards model-based integration of tools and techniques for embedded control system design, verification, and implementation. In *Models in Software Engineering: Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 – October 3, 2008. Reports and Revised Selected Papers*, 20–34. Springer-Verlag, Berlin, Heidelberg.

SAE International (2006). SAE architecture analysis and design language (AADL)(AS5506). `http://www.sae.org`

Whitehouse, K., Sharp, C., Brewer, E., and Culler, D. (2004). Hood: a neighborhood abstraction for sensor networks. In *Proceedings of MobiSys 2004*, 99–110. ACM, New York, NY, USA.