Using BIP to reinforce correctness of resource-constrained IoT applications

Alexios Lekidis *, Emmanouela Stachtiari †, Panagiotis Katsaros †, Marius Bozga * and Christos K. Georgiadis ‡
* Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
CNRS, VERIMAG, F-38000 Grenoble, France
email:firstname.lastname@imag.fr
† Department of Informatics, Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
email:{emmastac, katsaros}@csd.auth.gr
‡ Department of Applied Informatics, University of Macedonia
54006 Thessaloniki, Greece
email:{geor}@uom.edu.gr

***Abstract*—Internet of Things (IoT) systems process and respond to multiple (external) events, while performing computations for a Sense-Compute-Control (SCC) or a Sense-Only (SO) goal. Given the limitations of the interconnected resource-constrained devices, the execution environment can be based on an appropriate operating system for the IoT. The development effort can be reduced, when applications are built on top of RESTful web services, which can be shared and reused. However, the asynchronous communication between remote nodes is prone to event scheduling delays, which cannot be predicted and taken into account while programming the application. Long delays in message processing and communication, due to packet collisions, are avoided by carefully choosing the data transmission frequencies between the system's nodes. But even when specialized simulators are available, it is still a hard challenge to guarantee the functional and non-functional requirements at the application and system levels. In this article, we introduce a model-based rigorous analysis approach using the BIP component framework. We present a BIP model for IoT applications running on the Contiki OS. At the application level, we verify qualitative properties for service responsiveness requirements, whereas at the system level we can validate qualitative and quantitative properties using statistical model checking. We present results for an application scenario running on a distributed system infrastructure.**

I . Introduction

The main challenge in the design of systems for the Internet of Things (IoT) is to implement a lightweight architecture with abstractions for an appropriate execution environment, while staying within the resource limitations of the interconnected devices. Such an environment can be based on existing IoT operating systems ([1], [2], [3]), which facilitate system integration by abstracting hardware and allowing control of the system's nodes.

In this context, applications are implemented as *event-driven systems* with processes acting as *event handlers* that run to completion. Due to the resource limitations and under the condition that an event handler cannot block, all processes of a node share the same stack. When an event is destined for a process, the process is scheduled and the event - along with accompanying data - is delivered to the process through the activation of its event handler.

IoT operating systems decouple the applications' design from the low-level kernel functions, which provide CPU multiplexing and event scheduling. Thus, the development of IoT applications can proceed independently from their deployment, which has the advantages of programming at a higher-level, but opens a possibility for design errors at the overall system level. Depending on the way that IoT applications are eventually deployed in a distributed environment, they may have to handle and route many different types of events [4]. In general, it is hard to ensure seamless interactions between the system's components given their high heterogeneity (different device types, measurement units and interaction modes).

Application development often relies on loosely coupled web services based on REpresentational State Transfer (REST) that may be shared and reused. The REST architecture allows interconnected things to be represented as abstract resources controlled by a server process and identified by a Universal Resource Identifier (URI). We can thus access sensors over the web in a request/reply or publish/subscribe manner with ordinary web clients. The resources are decoupled from the services and can be represented by various formats (e.g. XML or JSON), while they are accessed and manipulated using the methods of the HTTP or the CoAP protocol. However, the use of web services is not straightforward, because many IoT applications require a multicast and asynchronous communication compared to the unicast and synchronous approach of typical Internet applications. Moreover, web services may have to reside in battery operated devices and in case of increased energy consumption their batteries cannot be easily changed when devices are deployed in inaccessible or distant areas.

Special purpose tools for testing by simulation are needed to aid the developers throughout the application design and deployment on the IoT system ([5], [6]). However, the currently available simulation techniques and tools are generally time consuming and hard to use. It is therefore a hard challenge to guarantee qualitative and quantitative correctness properties at the application and system levels. To this end, we advocate a model-based rigorous analysis approach using the BIP component framework [7]. BIP is a formal language that has been successful in building executable models of mixed software/hardware systems. BIP models can be formally analyzed for guaranteeing important functional correctness properties and for evaluating the system's performance. Validated models then support the systematic generation of code for deploying the modeled applications on a distributed system, thus preserving the verified properties in the actual implementation.

Our approach is based on the development of faithful models for the IoT application and system levels. A RESTful application is modeled and analyzed for deadlock freedom and other safety and liveness properties, whereas at the system level we validate important non-functional properties using

This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thalis Athens University of Economics and Business - SOFTWARE ENGINEERING RESEARCH PLATFORM.

statistical model checking. To the best of our knowledge, these analyses are not supported by the tools used in IoT application development. Our concrete contributions are as follows:

- A BIP application level model is introduced for RESTful IoT applications. The interconnected things are represented by abstract resources controlled by a server [8]. The resources are accessed and manipulated using the CoAP protocol in client/server request/responses.
- At the system level, we present a BIP model for applications running on the Contiki OS. This model was initially constructed according to the existing communication standards for the supported protocol stack and it was later calibrated based on software-dependent runtime constraints of the Contiki OS.
- A RESTful service-based application scenario is analyzed with state space exploration and statistical model checking. We provide results for key functional and non-functional requirements.
- We provide proof-of-concept results for studying error behaviors in the IoT system.

The rest of the paper develops along the following lines. Section II presents the current state of practice for the design of IoT applications. We specifically focus on the widely used Contiki OS, which is a typical case of the system architecture principles implemented in most IoT operating systems. Section III provides the necessary background on the BIP language and the supported statistical model checking functionality. Section IV describes in detail our BIP application and system level models. Section V presents the case study scenarios and the obtained state space exploration and statistical model checking results, including those illustrating the study of various error behaviors. Finally, the paper concludes with a critical view of our work's contributions and a discussion on the future research and development prospects.

II . IoT APPLICATION DESIGN

We focus on the specific programming abstractions of the Contiki OS, which in general adhere to the principles behind the programming models of today's IoT operating systems. However, our rigorous analysis approach is still applicable for other operating systems, if there are faithful models of their implemented programming abstractions[1].

The Contiki OS was initially introduced to support the development of Wireless Sensor Network (WSN) applications [1], but it was later extended, in order to be integrated into the IoT. Two kinds of events are handled: (i) *asynchronous events* which are enqueued by the kernel and are dispatched later to the target process, and (ii) *synchronous events* that cause the target process to be scheduled immediately; execution control returns to the event posting process only when the target process has finished the event processing. A *polling* mechanism is also provided, which is usually used by processes that check for status updates of hardware devices. Polling is realized by scheduling high-priority events, which trigger calls of all processes having a *poll handler*, in the order of their priority.

An event *scheduler* dispatches events to running processes and periodically calls processes poll handlers. Once an event has been scheduled, its event handler must run to completion since it cannot be preempted by the kernel.

[1]The Contiki OS is an open source software and its design is transparent to the development community.

```
1  PROCESS(client, ``Client Example'');
2  AUTOSTART_PROCESSES(&client);
3
4  static struct uip_udp_conn *conn;
5  static struct etimer et;
6  const int period = 160; /* in sec. */
7
8  PROCESS_THREAD(client, ev, data) {
9        PROCESS_EXITHANDLER(conn_close();)
10       PROCESS_POLLHANDLER(printf(``Process polled\n'');)
11       PROCESS_BEGIN();
12       SENSORS_ACTIVATE(button_sensor);
13       server1 = conn_new(); /* function call */
14       etimer_set(&et, period);
15       while(1) {
16         PROCESS_YIELD();
17         if (ev == sensors_event && data == &button_sensor)
18         { PROCESS_WAIT_EVENT_UNTIL
19              (ev == PROCESS_EVENT_TIMER);
20              . . . . .
21              /* function call for data request */
22              etimer_reset(&et);
23         } else if (ev == tcpip_event) {
24              . . . . .
25              /* function call to handle received data */
26         }
27       }
28       PROCESS_END();
29 }
```

Listing 1: Contiki client process

In typical Contiki applications, processes are implemented as lightweight threads, called *protothreads*, that do not have their own stack. A protothread consists of a C function and a single local continuation, i.e. a low-level mechanism to save and restore the context, when a blocking operation is invoked. Thus, when an event handler does not run to completion, the scheduling of other processes takes place. The protothread's local continuation is set before each conditional blocking wait. If the wait is to be performed, an explicit return statement is executed and the control returns to the caller. Upon the next invocation of the protothread the local continuation that was previously set is resumed, the program jumps to the same blocking wait statement and re-evaluates the condition. The protothread's execution continues, once it is allowed by the blocking condition. A local continuation is a snapshot of the current process state and its main difference from ordinary continuations is that the call history and values of local variables are not preserved. If some variables need to be saved across a blocking statement, this limitation can be sidestepped by declaring them as *static* local variables.

A typical Contiki application consists of processes defined in a C module. Listing 1 shows the code structure of a client process. In the `PROCESS` macro, the reference variable `client` and a string are assigned to the process, with the latter used in debugging. The `AUTOSTART_PROCESSES` macro (line 2) requests the process to be automatically started when the module is booted. The process code is enclosed in a `PROCESS_THREAD` macro, which defines the accessed variables, the handled event (`ev`) and its data. The process control flow is included between the two macros `PROCESS_BEGIN` and `PROCESS_END`, but handlers for the exit and poll events will be enabled independently from the control flow and are therefore placed before `PROCESS_BEGIN`.

In the infinite loop (line 15), the process waits for and processes events. First, it is blocked on line 16 until the receipt of an event that triggers evaluation of conditions in lines 17 and 22. Depending on whether the event comes from the button sensor or the network stack, the execution flow is diverted

accordingly. If both conditions are false, the process execution returns to the PROCESS_YIELD macro and it is blocked. The process is also blocked on the macro of line 18, until a timer fires an event. Then lines 20 and 21 are executed and the process is blocked again on line 15. Finally, upon the arrival of an exit event, the exit event handler is executed and the process reaches the PROCESS_END macro. Process execution ends when the PROCESS_END macro is reached.

In the Contiki OS, it is possible to develop applications based on loosely coupled RESTful web services that can be shared and reused. Listing 2 shows the code structure of a REST Contiki server comprising resource definitions, and a process for activating the REST resources and the REST engine (a number mediators between the application and communication layers are available for handling the dispatch of incoming messages and the delivery of their responses).

```
1  /* Getter. Returns the reading from light sensor */
2  RESOURCE(light, METHOD_GET, ''light'');
3  void light_handler(REQUEST* request, RESPONSE* response)
4  { uint16_t light_solar;
5    read_light_sensor(&light_solar); /* function call */
6    sprintf(temp,''%u;%u'',light_photosynthetic,light_solar);
7    rest_set_header_content_type(response, TEXT_PLAIN);
8    rest_set_response_payload(response, temp, strlen(temp));
9  }
10 /* Actuator. Toggles the red led */
11 RESOURCE( leds_toggle,
12   METHOD_GET | METHOD_PUT | METHOD_POST,
13  ''toggle'');
14 void leds_toggle_handler(REQUEST* req, RESPONSE* resp)
15 { leds_toggle(LEDS_RED); /* function call */ }
16
17 PROCESS(server, ''Rest Server Example'');
18 AUTOSTART_PROCESSES(&server);
19 PROCESS_THREAD(server, ev, data)
20 {
21   PROCESS_BEGIN();
22   /* Start rest engine process: handles
23      invocation of resource handlers */
24   rest_init();
25
26   SENSORS_ACTIVATE(light_sensor);
27   rest_activate_resource(&resource_light);
28   rest_activate_resource(&resource_toggle);
29   PROCESS_END();
30 }
```

Listing 2: RESTful Contiki server

A resource definition includes a resource variable, the supported HTTP methods and a URL. Our example shows the definitions of two resources, a light sensor (line 2) and a LED toggle switch actuator (line 11). A function conventionally named [resource_name]_handler is associated with each resource. This function is invoked upon every single URL request of the resource with one of the supported HTTP methods. The resource handlers enclose C code without protothread macros, since they do not implement independent processes. The light sensor resource handler of our example is a getter that reads the solar light indication and adds it to the response. On the other hand, the leds toggle handler switches the red LEDs on and off. The server process in line 19 initializes the REST engine process and activates the defined resources, in order to render them accessible. The REST engine implements the receipt of URL requests, the invocation of the appropriate handlers and the delivery of responses. Only one request is handled at a time and therefore the concurrent execution of resource handlers is not possible. However, another process implementation could be used instead of the REST engine.

The programming of Contiki applications can proceed independently from their deployment over the IoT system nodes. The asynchronous communication between remote nodes is prone to event scheduling delays, due to the execution of other processes on the same nodes. These delays cannot be predicted and taken into account, while programming the application. The problem is further complicated because all processes in a Contiki node share a single buffer for communication processing. The delays for the encoding and decoding of messages are hard to be predicted, since they depend on the node's overall CPU load. Moreover, there is high probability of a collision occurrence, if two or more nodes access the communication medium simultaneously. In this case, all the involved nodes will back off the transmission for a random period of time and will retry once this period has elapsed. To avoid long delays in message processing and communication, the developers will have to carefully set the data transmission frequencies between the system's nodes.

Currently, the only way to cope with the discussed problems is the analysis by simulation. For this purpose, the Contiki developers have introduced Cooja [9], a flexible Java-based simulation tool. Cooja is based on the lower level MPSim platform emulator [6] that provides accurate information for a system's underlying hardware. It is thus possible to investigate the system's functional and non-functional behavior and to inspect various performance aspects based on Contiki's native TimeLine module (e.g. message buffer utilization, energy consumption etc.). In the Cooja simulator, the user can select parameters for the network environment. Then, the IoT application's processes are allocated on the system's node representations, as imposed by the distribution of the interconnected devices. The functional behavior of the simulated IoT system can be tested through the simulation of simple execution scenarios. By simulating more realistic workloads, it is then possible to inspect the system's performance.

However, a system simulation can only provide partial assurance of the system's behavior and it is not adequate for guaranteeing correctness properties for and application's functional and non-functional requirements.

III . Background

A. The BIP component framework

BIP is an expressive component framework that allows building complex, hierarchically structured models from atomic components characterized by their behavior and their interfaces. We provide the formal semantics of the BIP framework, as is defined in [7] and earlier articles.

BIP atomic components are transition systems extended with a set of ports and a set of variables. An atomic component C is defined as a tuple (Q, X, P, T), where Q is a set of control locations, X a set of variables, P a set of communication ports and T a set of transitions. Each transition τ is of the form (q, p, g, f, q') where $q, q' \in Q$ are control locations, $p \in P$ is a port, g is a guard and f is the update function of τ. g is a predicate defined over variables in X and f is a function (BIP can invoke functions written in C/C++) that computes new values for X according to their current values.

In order to compose a set of n atomic components $\{C_i = (Q_i, X_i, P_i, T_i)\}_{i=1}^{n}$, we assume that their respective sets of ports and variables are pairwise disjoint. We define the global set $P \stackrel{def}{=} \bigcup_{i=1}^{n} P_i$ of ports. An interaction a is a triple (P_a, G_a, F_a), where $P_a \subseteq P$ is a set of ports, G_a is a guard and F_a is a data transfer function. By definition, P_a contains at

most one port from each component. We denote $P_a = \{p_i\}_{i \in I}$ with $I \subseteq \{1 \dots n\}$ and $p_i \in P_i$. G_a and F_a are defined on the variables of participating components, that is $\bigcup_{i \in I} X_i$.

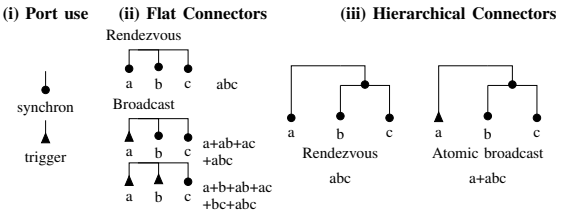

Fig. 1: Flat and hierarchical BIP connectors

Given a set γ of interactions, a priority is defined as a strict partial order $\pi \subseteq \gamma \times \gamma$. We write $a \pi b$ for $(a, b) \in \pi$, to express the fact that interaction a has lower priority than b.

A composite component $\pi\gamma(C_1, \dots, C_n)$ is defined by a set of components $C_1, \dots, C_n$, a set of interactions γ and a priority $\pi \subseteq \gamma \times \gamma$. If π is the empty relation, we may omit π and simply write $\gamma(C_1, \dots, C_n)$. A global state of $\pi\gamma(C_1, \dots, C_n)$ where $C_i = (Q_i, X_i, P_i, T_i)$ is defined by a pair (q, v), where $q = (q_1, \dots, q_n)$ is a tuple of control locations such that $q_i \in Q_i$ and $v = (v_1, \dots, v_n)$ is a tuple of valuations of variables such that $v_i \in Val(X_i) = \{\sigma : X_i \to \mathcal{D}\}$, for all $i = 1, \dots, n$ and for $\mathcal{D}$ being some universal data domain.

The behavior of a composite component $C = \gamma(C_1, \dots, C_n)$ is defined as a labelled transition system over the set S of global states of C and the transition relation with the following semantics: C can execute an interaction $a \in \gamma$, iff (i) for each port $p_i \in P_a$, the corresponding atomic component C_i allows a transition labelled by p_i (i.e. the corresponding guard g_i evaluates to true), and (ii) the guard G_a of the interaction evaluates to true. If these conditions hold true for an interaction a at state (q, v), then a is enabled at that state. Execution of a modifies participating components' variables by first applying the data transfer function F_a on variables of all interacting components and then the update function f_i for each interacting component. Components that do not participate in the interaction stay unchanged.

In BIP, interactions between components are specified by *connectors*. A connector defines a set of interactions based on the synchronization attributes of the connected ports (Fig. 1i), which may be either *trigger* or *synchron*:

- if all connected ports are synchrons, then synchronization is by *rendezvous*, i.e. the defined interaction may be executed only if all the connected components allow the transitions of those ports (Fig. 1ii),
- if a connector has one trigger, the synchronization is by *broadcast*, i.e. the interactions are all non-empty subsets of the connected ports with the trigger port (Fig. 1ii).

Connectors can export their ports for building hierarchies of connectors (Fig. 1iii). and can use data variables, in order to compute transfer functions associated with interactions. Computations take place iteratively either upwards (*up*) or downwards (*down*) through the connectors' hierarchy levels, but computed values are not stored between the execution of two interactions (connectors are stateless).

SBIP [10] is a BIP extension that relies on a stochastic semantics, for the verification of large-scale systems by using Statistical Model Checking ([11] [12]). Existing tool support allows verifying various forms of qualitative and quantitative properties connected to non-functional requirements.

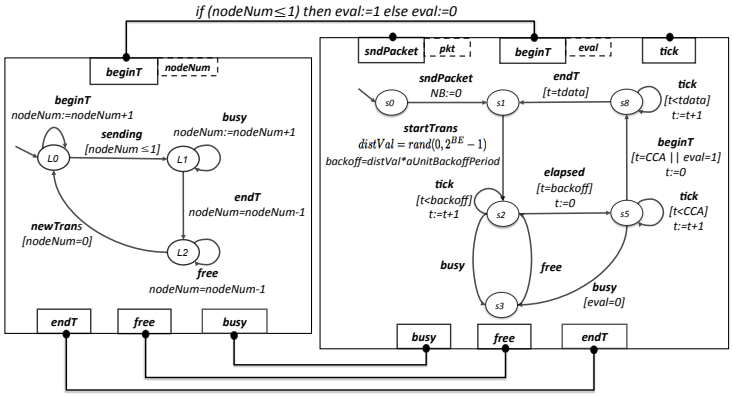

Fig. 2: Two atomic BIP components and their interactions

Figure 2 shows two atomic components from our IoT system model in BIP. On the left, we have the behavior of the communication *Channel* component that interacts through the ports *beginT*, *busy*, *free* and *endT* with the *ProtStack MsgSender* component on the right. The latter exhibits a stochastic behavior, due to the sampling from the uniform distribution for assigning value to the variable *distVal*.

B. Statistical Model Checking (SMC)

SMC was recently proposed as a means to cope with the scalability issues in numerical methods that are typically used to check stochastic systems. Consider a system model M and a set of non-functional requirements $R_1 \dots R_n$. Each requirement is formalized by a stochastic temporal property ϕ written in the Probabilistic Bounded Linear Temporal Logic (PBLTL) [10]. SMC is then used to apply a series of simulation-based analyses in order to answer two questions: (1) qualitative: is the probability $Pr_M(\phi)$ for M to satisfy ϕ greater or equal to a threshold θ? and (2) quantitative: what is the probability for M to satisfy ϕ? Both of these questions can serve to decide a PBLTL property.

Quantitative analysis aims to compute the value of $Pr_M(\phi)$. This depends on the existence of a counterexample to the negation of ϕ ($\neg\phi$), for the threshold θ. It is of polynomial complexity and, depending on the system model M and the property ϕ, it may or may not terminate within a finite number of steps. In [12], the authors propose an estimation procedure to compute a value for p', such that $|p' - p| < \delta$ with confidence $1 - \alpha$, where δ denotes the precision. This procedure is based on the Chernoff-Hoeffding bound [13].

Existing approaches for answering the qualitative question are based on hypothesis testing [11]. If $p = Pr_M(\phi)$, to decide if $p \geq \theta$, we can test *H*: $p \geq \theta$ against *K*: $p < \theta$. Such a solution does not guarantee a correct result but it allows to bound the error probability. The strength (α, β) of a test is determined by parameters α and β, such that the probability of accepting *K* (resp. *H*) when *H* (resp. *K*) holds is less or equal to α (resp. β). However, it is not possible for the two hypotheses to hold simultaneously and therefore the ideal performance of a test is not guaranteed. A solution to this problem is to relax the test by working with an indifference region (p_1, p_0) with $p_0 \geq p_1$ ($p_0 - p_1$ is the size of the region). In this context, we test the hypothesis $H_0 : p \geq p_0$ against $H_1 : p \leq p_1$ instead of *H* against *K*. If the value of p is between p_1 and p_0

(the indifference region), then we say that the probability is sufficiently close to θ, so that we are indifferent with respect to which of the two hypotheses K or H is accepted.

IV . RESTful application and Contiki system models

Our verification approach is based on a BIP model for the IoT system architecture. We currently support Contiki nodes, which are represented by BIP models at three levels (Fig. 3), namely, the REST module allocated to the node, the Contiki OS and the model for the protocol stack, which allows communication through the network channel. The BIP components for the REST modules comprise the *IoT application level model*, and the *IoT System Model* integrates them with the OS and protocol stack components. The interactions within and across Contiki nodes are represented by hierarchical rendezvous connectors.

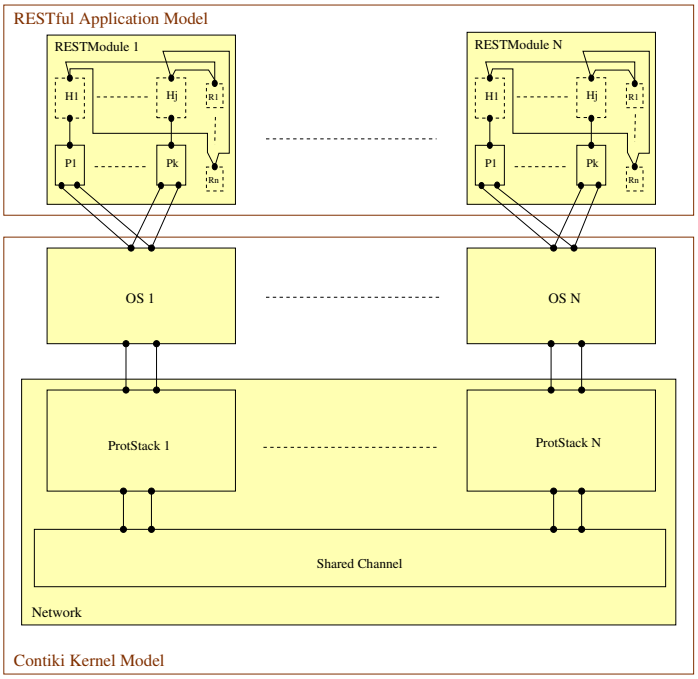

Fig. 3: BIP model for the Contiki IoT system architecture

In choosing the granularity of the BIP components behavior we opted to *faithfully model* all possible interleavings of events. Functional requirements verification takes place by state space exploration and can be extended to large systems with several client/server processes. The validation of non-functional requirements in large-scale systems requires appropriate stochastic abstractions for the IoT system model [14].

The timing aspect of the model depends on the granularity of a discrete time step, which has been defined based on the transmission time per bit through the Contiki protocol stack. This time is the inverse of the data rate of a Contiki network's access point. The smallest unit of data transmitted is one symbol (4 bits) and the symbol transmission time is:

$$symbolPeriod = \frac{4}{dataRate} \quad (1)$$

For an access point to a wireless communication medium that operates in a frequency at the 2.4 GHz band, the data rate is equal to 250 kbps, therefore from Equation 1 the symbol period is equal to $16\mu s$. Thus, our timing abstraction ignores delays smaller than the inverse of this data rate, which is $4\mu s$. However, this abstraction allows a much more fine-grained timing analysis compared to the one supported by the Cooja simulator, which is in the ms scale.

In our SBIP model, we have integrated values of important parameters, in order to model faithfully software-dependent runtime constraints of the Contiki OS. These parameters are: (i) the time needed for compression and decompression of the packets IP headers according to the HC1/HC2 encoding mechanisms [15], (ii) the pre and post buffering taking place for each packet transmission. The values for these parameters were obtained by measuring the duration from the beginning till the end of the corresponding executable code block within the Contiki OS. We note that the actual parameter values differ from system to system, since they mainly depend on the available computational resources.

A detailed definition of BIP model's structure (Figure 3) is given by the following grammar:

⟨*SystemModel*⟩ ::= ⟨*AppModel*⟩ ⟨*ConKernel*⟩
⟨*AppModel*⟩ ::= ⟨*RestModule*⟩+
⟨*RestModule*⟩ ::= ⟨*Process*⟩+ (⟨*Resource*⟩⟨*ResHandler*⟩)*
⟨*ConKernel*⟩ ::= ⟨*OS*⟩+ ⟨*Network*⟩
⟨*OS*⟩ ::= ⟨*Scheduler*⟩ ⟨*Timer*⟩ ⟨*CommHandler*⟩
⟨*Network*⟩ ::= ⟨*ProtStack*⟩+ ⟨*Channel*⟩
⟨*ProtStack*⟩ ::= ⟨*MsgSender*⟩ ⟨*MsgReceiver*⟩

The BIP model comprises two layers, namely the RESTful Application Model (AppModel) and the Contiki Kernel (ConKernel), which represent respectively, the application logic and the Contiki platform. The AppModel consists of several RestModule composite components that are deployed in different Contiki nodes. Each RestModule includes a number of process components (P1 to Pk) and optionally a set of resource components (R1 to Rn), representing REST resources, and resource handler components (Hi to Hj), which manipulate the resources. The ConKernel includes the composite components for the OS of every node and the Network composite component, with components for the protocol stack of each node and the communication channel.

A. RESTful Application Model

Fig. 4 shows the structure of the RestModule composite component and its interactions with the OS component. For simplicity, we show only the interactions of the OS with one process of the RESTModule component, but the same interactions are applied for several processes. Every process is represented by an atomic component, whose behavior depends on the application's logic.

Each component for a process interacts with the ConKernel model through the ports shown in Fig. 4. The process is called (*called* port) when an event destined for the process is scheduled. After handling the event, it yields execution (*yield* port). Synchronous or asynchronous events may be posted that are handled by other processes using the *postSyn* and *postAsyn* ports. If a synchronous event has been posted, the process resumes execution (*resume* port) upon the end of event handling. Each process may also request polling for itself or other processes. It can set deadlines using timers (*setTimer* port) or send a message (*sndMsg* port). ConKernel informs the process about the completion of setting a timer or transferring a message (*timerSet*, *msgSnt* ports). When a process finishes its execution the *end* port is enabled.

Fig. 5 shows the structure of a RestModule component with a server. The server process is represented by an atomic com-

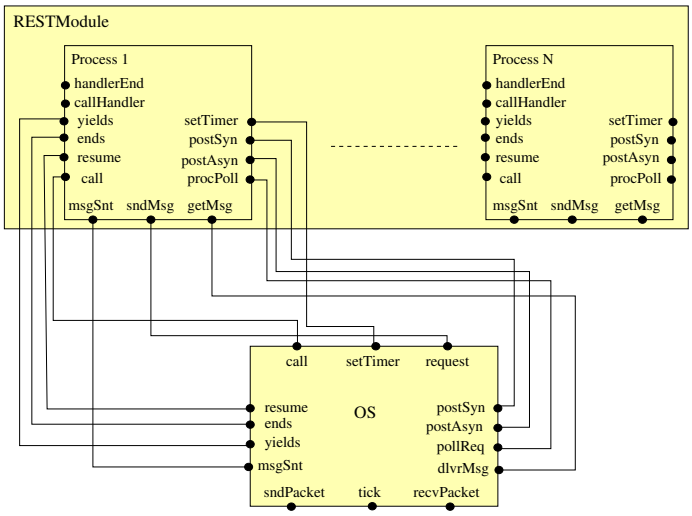

Fig. 4: The RestModule for a Client and its interactions with the OS

ponent with ports for calling a resource handler (*callHandler*) and for getting the response (*handlerEnd*). Every resource handler interacts with one resource, in order to submit a query (*sndQry* port) and get its result (*getRslt* port). Multiple resource handlers are allowed to interact with the same resource.

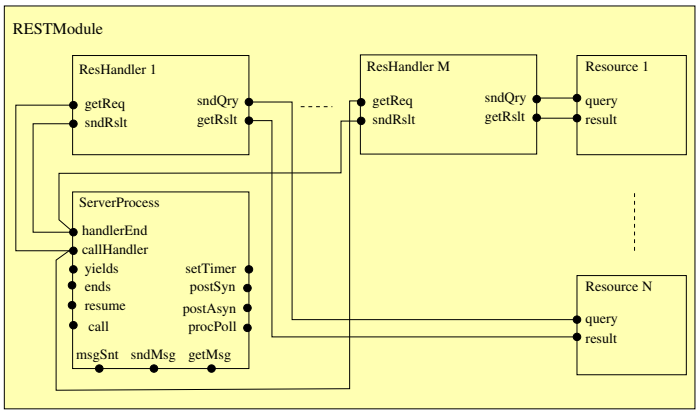

Fig. 5: The RestModule for a Server

B. Contiki Kernel Model

The ConKernel model consists of the OS and the Network composite components. The former models the behavior of the Contiki kernel [1] regarding the scheduling and the event-based interprocess communication. The latter represents the Contiki network module and therefore models the entire protocol stack.

As it is shown in Fig. 6, the OS composite component interacts with (i) the RESTModule component, in order to receive and handle incoming events and (ii) the Network component for the data exchange events. It consists of the Scheduler, the Timer and the CommHandler atomic components. The Scheduler component manages the incoming events from the processes of the RESTModule component. We distinguish four types of events, namely: (1) initialization (*INIT* event); (2) (a)synchronous event posting; (3) polling (*POLL* event); (4) yielding (*YIELD* event); and (5) exiting (*EXIT* or *EXITED* events). Events may be either synchronous and asynchronous, and are stored respectively in a stack and in a FIFO queue. The Scheduler checks periodically for the presence of incoming events, in the event queues (period T=$p_{scheduler}$). In sensor

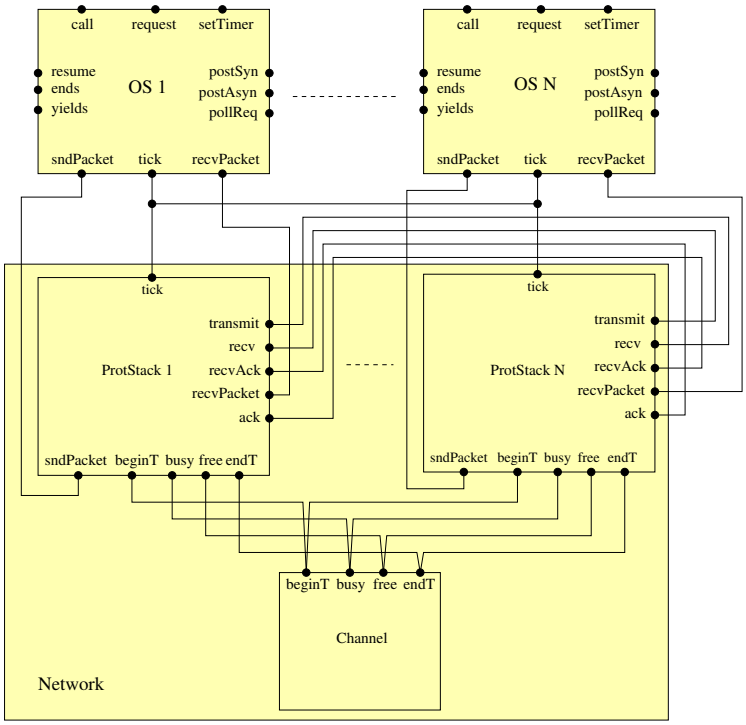

Fig. 6: Contiki Kernel Model interactions

applications, this period affects the the sensors' battery life: if the system operates in battery-saving mode, it wakes up to check for new events, when the period expires. The presence of an event, either triggers immediate handling - as for the synchronous events (*postSyn* port) - or the event is deferred - as for the asynchronous events (*postAsyn* port). The Timer component models a timer management process within the Contiki OS. It receives the incoming timer requests (*setTimer* port) of each RESTModule process and stores them as well in a stack. Each timer request includes a timer mode allowing to set, reset, restart or stop a particular timer. Finally, the CommHandler component models the TCP/IP process of the Contiki OS. It interacts with the Network component to either transmit the incoming packets or deliver the received packets to the process components of the RESTModule component. Thus, it uses one transmission and one reception buffer, called *TxBuffer* and *RxBuffer* respectively.

The Network composite component is comprised by the ProtStack and the Channel components. ProtStack consists of the *MsgSender* and *MsgReceiver* atomic components. The former models data transmission and the latter data reception. Both components represent data exchange through application layer messages of the CoAP or the HTTP protocol. Nevertheless, the transmission/reception mechanisms are handled by the underlying 6LoWPAN protocol of the Contiki protocol stack [16]. According to this protocol, the model uses the unslotted CSMA/CA mode of the IEEE 802.15.4 standard [17] (in the MAC layer) and applies an adaptation layer, in order to transport IPv6 packets over a Wireless Personal Network (WPAN). The Channel atomic component models the behavior of a communication medium, in order to receive requests for data transmission from network nodes, provide information regarding pending network transmissions, as well as resolve deterministically collisions that may arise from the simultane-

ous transmission attempt of multiple nodes. Fig. 2 illustrates the behavior and interactions of an abstract model for the ProtStack MsgSender and the Channel components. The Channel receives requests for data transmission through the interaction involving *beginT* ports of the two components. It approves one request for data transmission at a time and accordingly moves to the transmitting state L1 through the *sending* port. As long as it remains in L1, any following data transmissions from the other ProtStack MsgSender components will be halted through an interaction involving their *busy* ports. When the ongoing transmission ends, both components interact through the *endT* port and subsequently the Channel notifies all the components who requested access for data transmission that they can repeat their request (*free* port).

Model parameter	Value
aUnitBackoffPeriod	$20 * symbolPeriod$
CCA duration	$8 * symbolPeriod$
macMaxCSMABackoffs	0-5 (default 4)
macAckWaitDuration	$54 * symbolPeriod$
macMinBE	3
macMaxBE	3-8 (default 5)
aMaxFrameRetries	3
t_{data}	$[152, 1064] * symbolPeriod$
t_{ack}	$136 * symbolPeriod$
aTurnaroundTime	$12 * symbolPeriod$
SIFS	$12 * symbolPeriod$
LIFS	$40 * symbolPeriod$

TABLE I: Parameters of the modeled protocol stack

ConKernel is parametrized through the Network component. We distinguish two types of model parameters (Table I). Those who are fixed and those which can be modified, in order to analyze performance aspects of the protocol stack. In particular, the second type of parameters may concern the exponential backoff mechanism of the IEEE 802.15.4 standard (used in the MAC layer of the protocol stack) or the timeout for packet reception. The network throughput, as well as the number of collisions in the channel are strongly influenced by such parameters depending on the deployed application. Example parameters of the second type are the *macMinBE, macMaxBE, macMaxCSMABackoffs* and the *macMaxFrameRetries*.

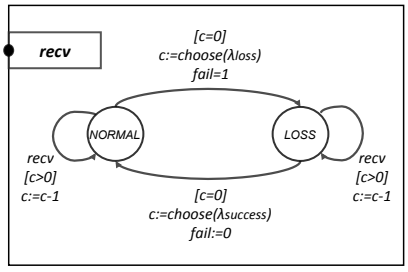

Fig. 7: FaultHandler component

C. Fault Model

We experimented with a fault model for injecting extensive bandwidth loss behavior. This allowed us to analyze the fault's impact on the tested application scenario of Section V. A high-level view of the injected behavior is shown by the FaultHandler component in Fig. 7. We distinguish the *NORMAL* and *LOSS* states, which represent respectively the successfully transmitted and the delayed or lost packets. FaultHandler receives all transmitted packets through the *recv* port and decides based on its state, if they will be delivered to their destination. It remains in each state, as long as the number of consecutive successful or delayed/lost packet transmissions is positive. This number is chosen by two probabilistic distributions, $\lambda_{success}$ and λ_{loss}, obtained from the analysis of debugging traces of a deployment on dedicated hardware platforms [18].

V . Case study

We introduce a SO smart heating IoT application that involves two subsystems, the home automation and the remote management. Home automation consists of a zone controller receiving temperature readings from sensor devices in the different rooms. For remote management, the zone controller is periodically accessed by a smartphone or tablet device through a Wide Area Network (WAN); control of the indoor heating and statistic records/profiles for the rooms are provided to the residents. In this scenario, we focus on the home automation subsystem that consists of one client, representing the zone controller and multiple REST server nodes for the temperature sensors, which communicate using the CoAP protocol. Let us consider an apartment with two rooms with a temperature sensor in each of them. The client periodically sends unicast GET requests sequentially to the two servers, which process them and reply. Responses are received within a certain time frame, before the client resends the request. Each node also transmits CoAP acknowledgments to signal a message receipt.

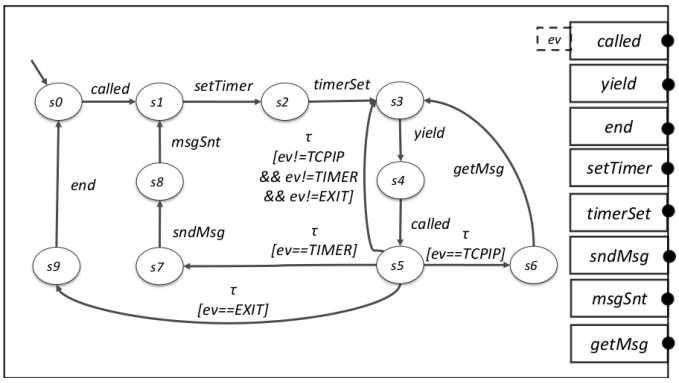

Fig. 8: Case-study client process

Fig. 8 shows the client process component for our application scenario. The process initially sets a timer and yields. The choice of this timer during application development is a trade-off between the freshness of the data and a potential overload of the network, leading to increased packet collisions. Whenever the client process is called upon a TIMER event, it sends a request, resets the timer and yields. When it gets called upon a TCP/IP event, it receives the response and yields. If it is called with an EXIT event its execution is ended.

The BIP model for the discussed application scenario consists of 8 atomic components for the RESTful Application Model and 16 atomic components for the Contiki Kernel Model. The model includes in total 147 connectors and 428 transitions and consists of 4100 lines of BIP code. Nevertheless, if a larger number of client and server nodes is assumed, the model can be easily scaled by properly adjusting the queue sizes and the state representation.

A. Verification of functional requirements

As mentioned in Section II, the development of Contiki applications is an error-prone procedure. Scheduling multiple processes may cause unpredictable delays in their event consumption. A client process can set a deadline for an expected response, or else resends the request. The deadline will have to be tuned such that it is feasible for the process to receive the response, in order to avoid sending redundant requests.

The analyzed functional and non-functional requirements are relevant for most IoT applications that involve a number of client and server processes. The functional requirements are:

FR1 The client eventually runs to completion (reaches `PROCESS_END`).

FR2 The client never sends redundant request messages to a server.

FR3 The node eventually transmits the requests sent by the client process.

FR4 The client eventually receives responses for the requests transmitted by its node.

FR5 All messages for a client transmitted in its node are eventually received by the client process.

FR1 is a consequence of the fact that our system model terminates, since all clients must terminate before the system does. For FR2, we consider a client request as redundant, if it causes the same response with the one for a previous request. Every such response may or may not reach the client process and this is monitored by the observer automaton in Fig. 9. FR3 requires that the client will transmit successfully all the sent requests. FR4 requires that the client will eventually obtain a response for each transmitted request. Finally, FR5 specifies that if a response has been received by the client's node, the client process will obtain it. For each of the requirements FR3 to FR5 an observer automaton property was derived.

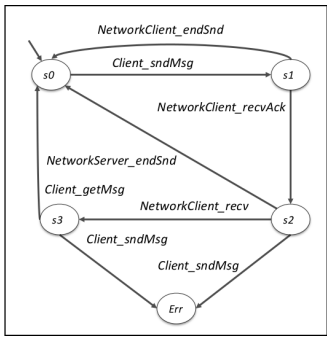

Fig. 9: Observer for property 2

The observer for the FR2 (Fig. 9) starts its execution when the client process sends a message (*sndMsg* port). After having received an ack (*recvAck* port), the observer waits in state $s2$. If the transfer of the request or the response fails (*endSnd* port), the observer returns to its initial state. Alternatively, if the process sends a new message before having obtained a response (*getMsg* port), which has arrived at the node (*recv* port) or before the server has completed the response dispatch (*endSnd* port), then state Err is reached respectively from $s3$ or $s2$, and the property is falsified.

We tested many different values for the client's timer, and we present the results for three: 14.5, 17.5 and 240 ms. The 14.5 timer was chosen randomly, and did not satisfy any of the requirements. In order to find a more appropriate timer, we used an observer automaton for each message transfer, which measured the time elapsed between the sending and the receiving of the response by the process. By choosing a timer higher than this value, i.e. the 17.5 timer, we had FR2 and FR5 being satisfied. However, FR3 was still violated for this timer, because a small period for sending client requests, incurs the possibility that the node might not be able to transfer a request due to a collision. From verifying FR5, we can only infer that FR3 fails, because a server might fail to transfer the response, due to a collision. For the same reason, FR4 is also violated for the 17.5 ms timer. However, both FR3 and FR4 hold for the highest timer value, which allows for a safe margin between consecutive requests, in order to avoid collisions.

B. Analysis of non-functional requirements

The overall performance of an IoT application deployed in a distributed and resource-constrained environment is connected to key non-functional requirements, such as the following:

NFR1 Memory saving by properly sizing the message buffers in each system node. Such buffers are used in Contiki for the communication through the protocol stack.

NFR2 Avoidance of overflow in the asynchronous event queue (FIFO) of each node.

NFR3 Relatively low collision rate in the communication medium, in order to avoid large communication latencies, which have a strong impact to the network performance and may increase the probability of packet losses.

C. Experiments

We conducted two sets of experiments. First, we compared the performance of the Contiki code for our application, when it is simulated in the Cooja environment with our IoT System Model in SBIP. We focused on the response time for the reply of a Server node to the Client's request, i.e. the total time elapsed for the end-to-end transmission, from the initialization of the message in the Server until it is reliably received by the Client. As shown by the sample window of Figure 10 the obtained results were similar, however the SBIP System Model had a slightly larger variability providing greater or smaller values than the observed outliers of the Cooja simulator. This is due to its improved accuracy, which enables a more fine-grained simulation compared to the Cooja environment.

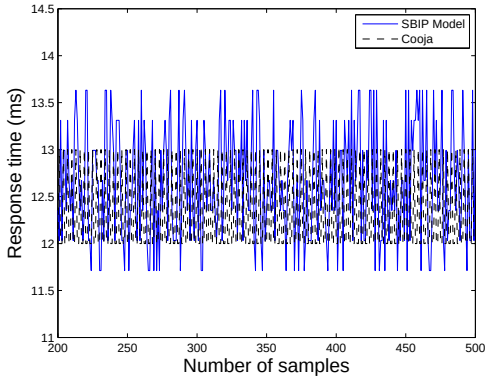

Fig. 10: SBIP/Cooja response times for each Server (in ms)

The second set of experiments concerned with the injection of a realistic representation of bandwidth loss in the IoT system model through probabilistic distributions, which results in consecutive packet losses or out-of-order delivery in the system. The main reason for analyzing packet losses is that when a network node transmits a packet, it waits for the acknowledgment in a certain time frame, namely the *macAckWaitDuration* parameter of the MAC layer (provided in Table I), before trying a retransmission. If this time expires, the following retransmission will increase the packet transmission frequency, leading eventually in higher probability of collisions in the system. Indeed, Figure 11 illustrates such a behavior obtained from the simulation of the SBIP model with the addition of the Fault Model in BIP, presented in Section IV.

In particular, the response time for the reply of Server 1 to the Client's request can be increased up to 23.8 ms, due the presence of collisions in the communication medium. Injection of packet losses is also possible in the Cooja simulator, through its UDGM - Distance Loss mode. However, since this behavior is based on user-provided simulation values for parameters as the *SUCCESS_RATIO_TX* and *SUCCESS_RATIO_RX*, it cannot reflect the reality accurately. Therefore, our analysis exceeds the simulation capabilities of the Cooja environment.

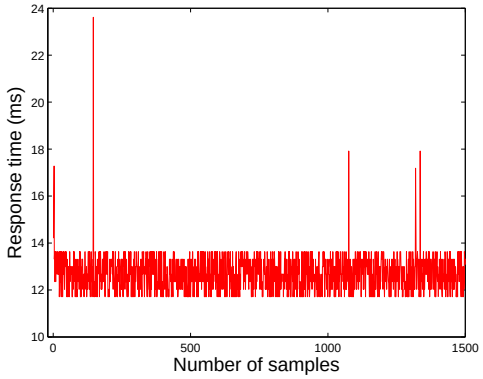

Fig. 11: SBIP response times for each Server (in ms) with fault injection

The construction of the SBIP System Model also allowed the analysis and validation of the aforementioned non-functional requirements for the case study. Therefore, we accordingly describe them with stochastic temporal properties using the Probabilistic Bounded Linear Temporal Logic (PBLTL) formalism [10] and present their evaluation results through the SBIP model checking tool.

NFR1. In order to satisfy this requirement several properties can be considered. Such properties are 1) $\phi_1 = (size(TxBuffer)) < A$ and 2) $\phi_2 = (size(RxBuffer) < B)$, where $size(TxBuffer)$ and $size(RxBuffer)$ indicate the size of the transmission and reception buffer of the protocol stack respectively. B and A are fixed non-negative numbers representing a bound for the size of the two buffers. For the sake of brevity in this experiment we focus on ϕ_2, namely the reception buffer of the CommHandler component, however this analysis can be evenly conducted for the transmission buffer. In this experiment we tried to estimate the value of B as it varies according to the value of the period that the Scheduler uses to check for the presence of incoming events in the event queues ($p_{scheduler}$). This period determines the frequency with which the Scheduler posts events concerning the communication through the protocol stack to the CommHandler component. If this frequency is increased, the number of transmitted packets over the network is equally increased. Therefore, more packets are received in the reception buffer of the CommHandler component. In particular, we have experimented with different values for $p_{scheduler}$, such as $p_1 = 0.1ms$, $p_2 = 10ms$ and $p_3 = 1s$. For $p_{scheduler} = p_1$, as illustrated from Figure 12, $P(\phi_2) = 1$ for A equal to 1. If $p_{scheduler}$ is increased and is equal to p_2, $\phi_2 = 1$ for B equal to 5. In the worst-case scenario ($p3$), $p_{scheduler}$ was equal to the packet transmission period (1s in the specific case study) and the value of A has to be 10, in order to guarantee that the ϕ_2 always holds. The size of the reception buffer can be adjusted by the parameter *MAX_NUM_QUEUED_PACKETS* of the Contiki OS, found specifically in the core module of the Contiki kernel. This parameter corresponds to the value of A in our analysis and is initially equal to 2.

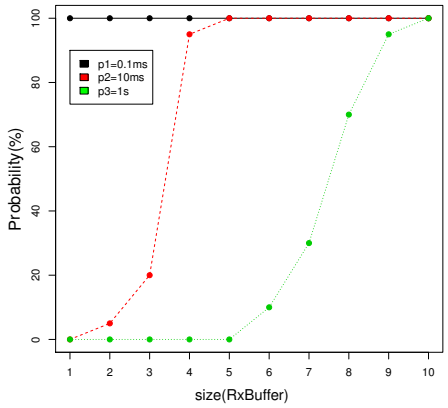

Fig. 12: Reception buffer size for different event scheduling periods

NFR2. We have accordingly expressed the second requirement as: $\phi_3 = (size(AsynFIFO) < MAX)$, where $size(AsynFIFO)$ indicates the size of the asynchronous event queue in the model. For this experiment we have considered the value of MAX equal to 10. As for the particular case study we only consider a limited number of asynchronous events, we have evaluated that this property holds always ($\phi_3 = 1$) for the chosen value of MAX.

NFR3. This requirement is expressed as the property: $\phi_4 = (NC \leq 1)$, where NC indicates the number of successive retransmissions following the occurrence of a collision in the model. In order to evaluate this property we have conducted two sets of experiments the first one only on the IoT System Model and the second with the addition of the Fault Model. We have tested the property ϕ_4 for the first experiment in a large number of communication cycles and evaluated it as $P(\phi_4) = 1$, meaning that no collisions are present in the communication medium. However, for the second experiment the impact the same property was evaluated as $P(\phi_4) = 0.55$, as the extensive loss of bandwidth increased the packet transmissions and hence the number of collisions in the SBIP model. This is also illustrated in Figure 11.

VI . Related work

An important direction of research is the Cloud centric vision for the IoT, which leads to its worldwide implementation. The Cloud receives data from ubiquitous sensors, analyzes it and provides the user with easy to understand web-based visualization. Developing IoT applications using low-level Cloud programming models and interfaces is complex. To overcome this, authors in [19] underline the need of an IoT application-specific framework for rapid creation of applications and their deployment on Cloud infrastructures.

A model-based methodology for the development of IoT applications in WSN has been presented in [20]. The proposed framework utilizes graphical Matlab tools to model and simulate the behavior of an application's components at a high-level. While being a promising approach which enables the generation of network simulations to be executed in Contiki, it does not address specific functional and non-functional requirements. DiaSuite [21] supports a framework for the different development phases of SCC applications, through an

integrated high-level specification. Although it is enriched with a methodology to address non-functional requirements, validation support for them is not provided. Additionally, verification techniques on functional requirements are not considered. An approach in model-based development, closer to ours is the modeling of system and application behaviors utilizing Finite State Machines (FSM). In [22], authors introduce an FSM approach to enable a model-driven development approach for service-based IoT applications. Through the utilization of FSM, the application logic can be described, which in sequence can be evaluated though a GUI tool.

Alternative WSN operating systems which were extended for IoT applications include the RESTful API implementation for TinyOS [3] and the Linux environment. However, considerable limitations exist for those two operating systems: Linux consists of a monolithic kernel, which lacks modularity and often results in a complex structure that is hard to understand, especially for large-scale systems. TinyOS on the other side, uses an event driven code style, thus it cannot support multithreading. An emerging under-development IoT operating system for such applications is RIOT [2], which is similar to Contiki and includes further optimizations in the resource usage. Nevertheless, as the development is still ongoing, it doesn't include an implementation for CoAP and porting of the OS to IoT platforms is not yet fully provided.

VII . Conclusion

We presented a rigorous model-based analysis for resource-constrained IoT applications using the BIP component framework. The whole approach was demonstrated through the systematic construction of faithful BIP models for RESTful service-based applications over nodes running the Contiki OS. All the developed models along with additional technical details are available online[2]. As a proof of concept, our approach was applied to a smart heating IoT application with a client and multiple server nodes. Important functional and non-functional requirements were verified, which extend the analysis possibilities beyond those provided by the Contiki simulation tools. On the other hand, we showed that simulation results from our BIP model are very similar to those obtained by the Contiki's Cooja simulator. Various aspects of service responsiveness were checked by state space exploration, whereas non-functional requirements concerning buffer utilization, collision rate and blocking time in the event queues were analyzed using statistical model checking. Finally, we proposed a fault injection approach for exploring the impact of various error-prone behaviors in the network communication.

For state space exploration, only a limited number of nodes and client/server processes is required that suffice to generate all the different execution traces at the node level. For being able to explore the performance aspects of an IoT application deployed on a large-scale system, we plan to develop an appropriate stochastic abstraction of the presented IoT system model [14]. Such an abstraction can be applied only to the OS and protocol stack models, in order to avoid loosing event interleavings due to the application's functional behavior. Furthermore, we consider possible extensions in the IoT system model, in order to analyze various security risks related to the DTLS transport and the CoAP protocols [23] or the HTTPS URI scheme [24], as well as their overall impact on the system's performance.

[2]http://depend.csd.auth.gr/ServiceSystemsModelling.php

References

[1] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *LCN'04*. IEEE, 2004, pp. 455–462.

[2] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, T. Schmidt *et al.*, "RIOT OS: Towards an OS for the Internet of Things," in *INFOCOM'13*, 2013.

[3] L. Schor, P. Sommer, and R. Wattenhofer, "Towards a zero-configuration wireless sensor network architecture for smart buildings," in *BuildSys'09*. ACM, 2009, pp. 31–36.

[4] A. Castellani, N. Bui, P. Casari, M. Rossi, Z. Shelby, and M. Zorzi, "Architecture and protocols for the Internet of Things: A case study," in *PERCOM Workshops*, March 2010, pp. 678–683.

[5] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo, "Declarative Tracepoints: A Programmable and Application Independent Debugging System for Wireless Sensor Networks," in *SenSys'08*. New York, NY, USA: ACM, 2008, pp. 85–98.

[6] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón, "COOJA/MSPSim: interoperability testing for wireless sensor networks," in *SIMUTools'09*, 2009, p. 27.

[7] A. Basu, S. Bensalem, M. Bozga, P. Bourgos, M. Maheshwari, and J. Sifakis, "Component Assemblies in the Context of Manycore," in *LNCS'13*. Springer, 2013, vol. 7542, pp. 314–333.

[8] E. Stachtiari, N. Vesyropoulos, G. Kourouleas, C. K. Georgiadis, and P. Katsaros, "Correct-by-Construction Web Service Architecture," in *SOSE'14*. IEEE, 2014.

[9] F. Österlind, "A sensor network simulator for the Contiki OS," *SICS Research Report*, 2006.

[10] S. Bensalem, M. Bozga, B. Delahaye, C. Jegourel, A. Legay, and A. Nouri, "Statistical Model Checking QoS properties of Systems with SBIP," in *ISOLA'12*. Springer, 2012, pp. 327–341.

[11] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *Runtime Verification*. Springer, 2010, pp. 122–135.

[12] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet, "Approximate probabilistic model checking," in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2004, pp. 73–84.

[13] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American statistical association*, vol. 58, no. 301, pp. 13–30, 1963.

[14] A. Basu, S. Bensalem, M. Bozga, B. Delahaye, and A. Legay, "Statistical abstraction and model-checking of large heterogeneous systems," *STTT'12*, vol. 14, no. 1, pp. 53–72, 2012.

[15] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 packets over IEEE 802.15. 4 networks," *RFC*, vol. 4944, 2007.

[16] Z. Shelby and C. Bormann, *6LoWPAN: The wireless embedded Internet*. John Wiley & Sons, 2011, vol. 43.

[17] L. S. Committee *et al.*, "Part 15.4: wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (LR-WPANs)," *IEEE*, 2003.

[18] A. Lekidis, P. Bourgos, S. Djoko-Djoko, M. Bozga, and S. Bensalem, "Building Distributed Sensor Network Applications using BIP," in *SAS'15*. IEEE, 2015.

[19] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Gen. Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[20] Z. Song, M. T. Lazarescu, R. Tomasi, L. Lavagno, and M. A. Spirito, "High-Level Internet of Things Applications Development Using Wireless Sensor Networks," in *IoT*. Springer, 2014, pp. 75–109.

[21] B. Bertran, J. Bruneau, D. Cassou, N. Loriant, E. Balland, and C. Consel, "DiaSuite: A tool suite to develop Sense/Compute/Control applications," *Science of Computer Programming*, vol. 79, pp. 39–51, 2014.

[22] N. Glombitza, D. Pfisterer, and S. Fischer, "Using state machines for a model driven development of web service-based sensor network applications," in *ICSE'10*. ACM, 2010, pp. 2–7.

[23] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," 2014.

[24] E. Rescorla, "HTTP over TLS." IETF, 2000.