Model-based design of IoT systems with the BIP component framework

Alexios Lekidis, Emmanouela Stachtiari, Panagiotis Katsaros, Marius Bozga, Christos K. Georgiadis

Model-based Design of IoT Systems with the BIP Component Framework

May 16, 2024

Alexios Lekidis, Emmanouela Stachtiari, Panagiotis Katsaros
Department of Informatics, Aristotle University of Thessaloniki 54124 Thessaloniki, Greece
E-mail address: alekidis, emmastac, katsaros @csd.auth.gr

Marius Bozga
Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France CNRS, VERIMAG, F-38000 Grenoble, France
E-mail address: marius.bozga@imag.fr

Christos K. Georgiadis
Department of Applied Informatics, University of Macedonia 54006 Thessaloniki, Greece
E-mail address: geor@uom.edu.gr

Abstract

The design of software for networked systems with nodes running an Internet of Things (IoT) operating system faces important challenges, due to the heterogeneity of interacting things and the constraints stemming from the often limited amount of available resources. In this context, it is hard to build confidence that a design solution fulfills the application's requirements. This paper introduces a design flow for web service applications of the Representational State Transfer (REST) style that is based on a formal modeling language, the BIP (Behaviour, Interaction, Priority) component framework. The proposed flow applies the principles of separation of concerns in a component-based design process that supports the modular design and reuse of model artifacts. The BIP tools for state space exploration allow verifying qualitative properties for service responsiveness, i.e. the timely handling of events. Moreover, essential quantitative properties are validated through statistical model checking of a stochastic BIP model. All properties are preserved in actual implementation by ensuring that the deployed code is consistent with the validated model. We illustrate the design of a REST sense-compute-control application for a Wireless Personal Area Network architecture with nodes running the Contiki OS. The results validate qualitative and quantitative properties for the system and include the study of error behaviours.

Keywords: Internet of Things; Model-based design; Service Oriented Architecture

1 INTRODUCTION

The Internet Of Things (IoT) aims at the seamless interconnection of heterogeneous embedded systems using the Internet technologies and infrastructure. The connected *things* are network nodes equipped with low-memory and low-power devices that collect data from the surroundings (sensors) and communicate it to the system, as well as smart objects that perform computations. System integration is facilitated by operating systems [1, 2, 3], which enable the nodes' control by abstracting the provided hardware and software resources.

Many different programming models have been proposed for the various types of IoT applications [4, 5, 6, 7]. In practice, software design relies heavily on node programming close to the operating system level. This affects the development time, the application reliability and its efficiency, due to the heterogeneity of the involved things and the diverse interaction modes to be taken into account. For things with continuous communication, a fixed schedule with periodic transmissions will have to be designed. If the application includes event-driven and command-based communications, packet collisions and message overloading may be encountered. Moreover, applications are not easily adapted to the ever evolving needs, especially when they are deployed in large-scale distributed environments, where they handle and route many different types of events [8].

The design complexity of an IoT application is reduced significantly, when re-using web services [9, 10]. According to the REpresentational State Transfer (REST) style, the things are accessed as abstract resources located by Universal Resource Identifiers (URIs) and they are manipulated through the Hypertext Transfer Protocol (HTTP) or the Constrained Application Protocol (CoAP) [11]. However, IoT applications primarily involve asynchronous interactions as opposed to the synchronous interactions found in most Internet applications. Furthermore, a typical web service is a long-running executable process, whereas IoT applications are based on resource-constrained things, which may have to remain idle for relatively long periods of time.

To address the risks due to the overall design complexity, operating systems provide tools [12, 13] for the simulation-based validation of execution scenarios. This support is insufficient for guranteeing at design time requirements of applications and systems. To this end, a *model-based design process* relying on formal modeling semantics can open prospects for an exhaustive analysis of the application's behaviour, and a simulation analysis grounded on statistical confidence.

We therefore advocate a design flow based on the BIP component framework [14]. The system design is specified in a Domain Specific Language, which is used to preserve the consistency between the auto-generated BIP model and the application code. BIP is a language with formally defined semantics for building executable models of mixed software/hardware systems. With its expressive coordination primitives, BIP facilitates the modeling of heterogeneous computations and interactions (synchronous and asynchronous) that are inherent in service-based IoT applications [15, 16]. It also allows using resource variables that model resources (e.g. time, memory, energy). The BIP model is amenable to formal verification through state space exploration, for guaranteeing behavioural correctness properties. If the model is extended with stochastic variables, it is amenable to statistical model checking [17], a simulation-based analysis with statistical guarantees, i.e. finely controlled quality of results by various confidence parameters. This allows validating quantitative properties derived from requirements relevant to the IoT system's architecture [18].

Through analysis by state space exploration we ensure deadlock freedom and other properties related to handling events in a timely manner, i.e. what we call service responsiveness. With statistical model checking, we validate properties for the correct operation of the system under statistical assumptions for its external stimuli, as well as the message buffer utilisation, the collision occurrence and the event queue blocking times [19]. These analyses are not supported by today's IoT operating system simulators. The concrete contributions of this research work are as follows:

- We present our model-based design flow in the context of IoT WPAN (Wireless Personal Area Network) systems.
- The domain specific language (DSL) for REST applications running on the Contiki OS [1] is then introduced; the focus on the Contiki OS was motivated by the fact that it is open source and its design is transparent to the development community.
- We illustrate the design of a REST sense-compute-control (SCC) application for a building automation system. The BIP model was based on the WPAN architecture standards and it was subsequently calibrated based on the Contiki OS runtime constraints.
- We provide results for properties derived from requirements for IoT WPAN systems. Moreover, a study of the system's robustness with respect to error behaviours is presented.

The described approach supports a component-based design philosophy that allows locating design errors to specific components, while boosting modular design and reuse of model artifacts. Through the separation of concerns, it enables the application development independently from the IoT system architecture. Finally, based on the wide range of supported analyses, multiple requirement types can be validated. The BIP models for the Contiki OS were first presented in [20]. Compared to that work, we introduce here the design flow with the DSL, and we present a SCC use case (a sense-only application was shown in [20]). This allowed us to additionally provide statistical model checking results for the system's operation under assumptions for its external stimuli.

Section 2 provides the background on the foundations of IoT systems, and on the BIP tools. Section 3 introduces the model-based design flow. Section 4 presents the case study and Section 5 comments on the benefits and the limitations of our approach. A comparison with other design methods is also included. The paper concludes with a summary of our work's contributions.

2 BACKGROUND

2.1 Foundations of IoT Systems

In an IoT system, the interactions amongst the different abstraction layers of the mixed SW/HW architecture (Figure 1) impact the overall system's performance and efficiency. From another point of view, we often have different applications overlapping in networks of heterogeneous things. The REST design is an architectural style that enables application-layer interoperability, i.e. multiple applications can co-exist and share the same set of things. It is used in the design of node-decentralized systems, but the nodes may be also integrated in cloud-centric solutions [21], which are more often preferred. Here, we only consider the former case. In most IoT operating systems, there is integrated support for the design of REST applications.

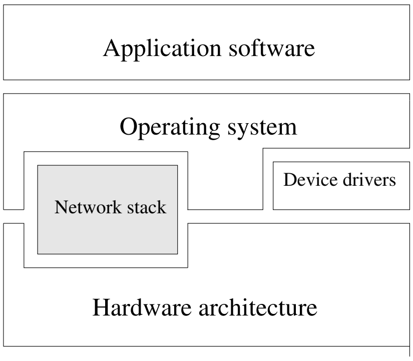


Figure 1: Typical SW/HW architecture of IoT node

Every message exchange involves a "server" node and one or many "clients". Clients send requests to the server and receive responses; they are responsible for keeping the state of the involved interactions. The servers accept requests in the form of CoAP/HTTP methods (e.g GET, POST, PUT, DELETE), for retrieving or modifying the state of their *resources*; every resource is uniquely identified by a URI and encapsulates data, such as thing descriptions and locations, sensor values or the state of an actuator. The same node may act as a server in some communications (e.g. to provide access to sensor values), and as a client in others (e.g. to register in a directory). There are also intermediary nodes, which implement both the client and server roles, but they only forward requests or translate them in other protocols. A gateway node may be used to mediate client requests and forward them to the actual server; such nodes often encapsulate legacy services, to improve server performance through caching, and to enable load balancing across multiple machines.

The IoT applications for WPAN systems can be classified in two broad categories according to how they manage their internal interactions and the interactions with the physical environment. The *Sense-Only* (SO) applications collect sensor data whenever necessary (intermittent sensing) or on a regular basis. Such examples are the smart heating systems, where one can remotely access and adjust the in-house temperature using cloud services, as in [20]. On the other hand, applications of the SCC category, apart from sensing, also coordinate various control activities. Such activities range from e.g. taking proactive actions for controlling the room's temperature to dispatching client notifications. The main difference between these two application categories is the autonomic operation of the SCC nodes, i.e. their functioning without any human intervention.

2.2 Contiki and REST application programming

The Contiki OS implements a lightweight architecture for *event-driven* applications [1]. The node processes wait for events and handle them in *event handlers* that run sequentially with respect to other handlers without being interrupted, i.e. they finish upon running to completion. This means that lengthy event handlers can absorb the processing capacity [22].

Processes communicate by posting *synchronous* and *asynchronous* events to each other. Synchronous events are dispatched immediately when they are posted, so that they are handled on the spot. The execution control returns to the calling process only after the event handler has finished. On the other hand, asynchronous events are en-queued by the OS and are dispatched later. While synchronous events target a single process, asynchronous events can target many processes. In that case, each process is called in a sequential manner.

Events are identified by their type. Along with the predefined event types by the OS, we can have application-specific types defined by processes. Two commonly used predefined events are the *poll* and the *exit* events. The poll events are asynchronous events of high priority; they are dispatched before any other asynchronous event. The exit events are posted synchronously and cause a process to run the exit handler and eventually end. As opposed to other handlers, the optional poll and exit handlers are enabled at all control flow locations of the process, though they might be omitted.

The Contiki OS features a REST Engine API [23], for the definition of REST service resources. A basic REST resource has a name, a URI-path and a set of HTTP/CoAP methods through which the resource is accessed. Each access is handled by a dedicated *resource handler* that responds to the requests coming from the network. A REST Engine process listens to the network stack and transfers the requests/responses from/to the resource handlers. The design of REST applications typically consists of the following steps:

1. The REST resource definitions are provided and the resource handlers are implemented or reused.
2. The application behaviour is distributed to nodes with the client, the server or both roles. The server processes, which activate the REST engine and the client processes with the methods to access server resources are implemented.
3. Appropriate parameters for the network are configured.
4. The functional behaviour is debugged using the OS's simulator.
5. When enough testing confidence is achieved, the client and REST server implementations are deployed on the system nodes.

The Cooja simulator [13] is used to validate the functional behaviour and when simulating realistic workloads the performance aspects of Contiki OS applications. However, such a simulation provides insight only for a limited set of execution scenarios and cannot ensure by itself the diverse requirements of an IoT system. Programming for resource-constrained devices involves various sources of delay that are hardy predictable. First, asynchronous communication within a node or between remote nodes is prone to delays, due to the execution of other processes and the fact that events are handled sequentially. Second, delays might occur due to message encoding and decoding, which depends on the overall CPU load of nodes. Finally, there is a high probability of collision, 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. With Cooja, the programmer is restricted to inspecting the behaviour of application functions and the performance of system nodes, without being able to inspect the interactions at the lower layers of the node architecture shown in Figure 1.

2.3 On the Tools of the BIP methodology

2.3.1 The BIP component framework

BIP (Behaviour-Interaction-Priority) [24] is a formal framework for building complex systems by coordinating the behaviour of a set of atomic components. *Behaviour* is defined as a transition system extended with data and C/C++ functions. The definition of coordination between components is layered: in the first layer lie the component *interactions*, while the second layer involves dynamic *priorities* between interactions.

The atomic components are finite-state automata extended with variables and ports. Variables are used to store local data. Ports are named, and may be associated with variables. They are used for interaction with other components. States denote control locations at which the components await for interaction. A transition is an execution step, labelled by a port, from a control location to another. It might be associated with a boolean condition (guard) and a computation defined on local variables. The model's global state at each execution step is given as the current control locations and the values of local variables of all atomic components.

Connectors relate ports from different subcomponents. by assigning to them a synchronization attribute: trigger (▲) or synchron (●). The connectors represent sets of interactions, that are, non-empty sets of ports which are jointly executed. If all connected ports are synchrons, one interaction is executed, that is, if all connected components allow the transitions of those ports (rendezvous). If a connector has one trigger, the possible interactions include at least one trigger port (broadcast). For every interaction, the connector might provide a guard and a data transfer, that are, respectively, an enabling condition and an exchange of data across the ports involved in the interaction. Additionally,

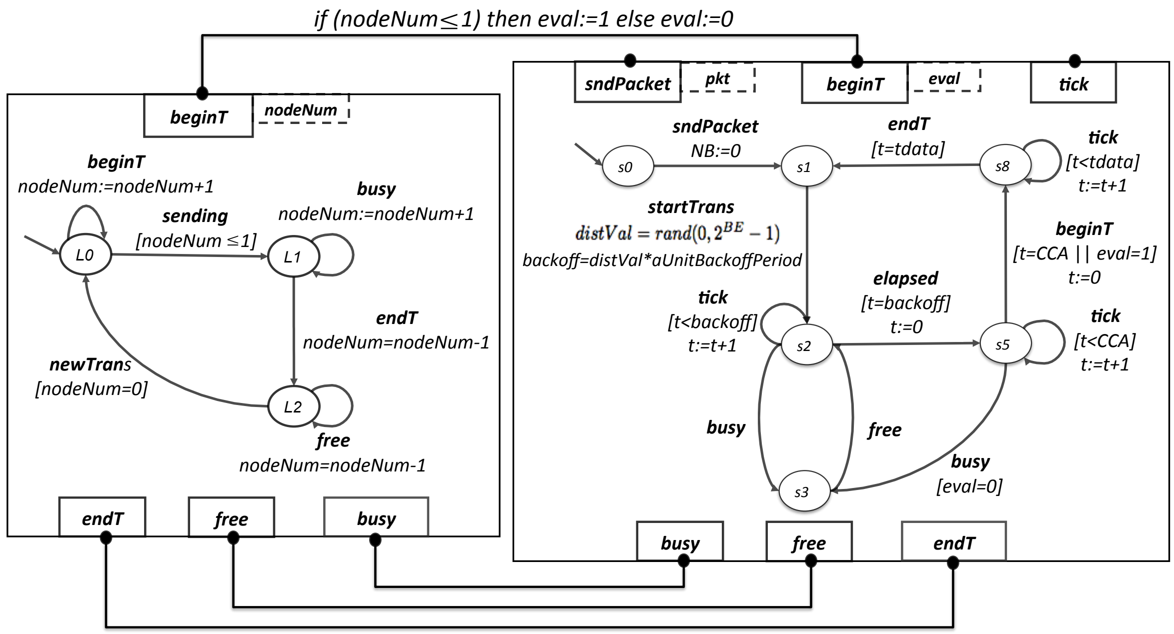


Figure 2: Example BIP components and their interactions - Channel (left) and ProtStack.MsgSender (right)

connectors can export ports for building hierarchies of connectors. The stochastic extension of BIP [25] allows (1) to specify stochastic aspects of individual components and (2) to provide a purely stochastic semantics for the parallel composition of components through interactions and priorities. Syntactically, stochastic behaviour at the level of atomic BIP components is obtained by using probabilistic variables. These are attached to probability distributions (implemented as C functions) and are updated (or sampled) during transition firing where they get random values accordingly. The semantics on transitions is thus fully stochastic. The stochastic semantics also covers the interaction level. When several interactions are enabled after application of priority rules, a probabilistic choice among them is performed using a user-specified probability distribution.

Figure 2 shows the composition of two BIP components from our IoT system model. On the left, the *Channel* models access to a shared communication medium. This component interacts through the ports *beginT*, *busy*, *free* and *endT*. On the right, the *ProtStack.MsgSender* component models the sending of data packets through the network and exhibits a stochastic behaviour, due to sampling from the uniform distribution for assigning a value to the *distVal* variable in transition *startTrans*. Only strict synchronization with synchrons (rendezvous) is used for these two components.

2.3.2 Statistical Model Checking

Statistical Model Checking (SMC) was proposed as a means to cope with the scalability issues in numerical methods for the analysis of stochastic systems. Consider a system model M and a set of requirements, where each requirement can be formalised by a stochastic temporal property ϕ written in the Probabilistic Bounded Linear Temporal Logic (PBLTL) [26]. The SMC applies a series of simulation-based analyses to decide PBLTL properties of the following two types:

1. *Is the probability $Pr_M(\phi)$ for M to satisfy ϕ greater or equal to a threshold θ?* Existing approaches to answer this question are based on hypothesis testing [27]. When $p = Pr_M(\phi)$, 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 of a test is determined by the parameters (α, β), such that the probability of accepting K (resp. H) when H (resp. K) holds is less than 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.

2. *What is the probability for M to satisfy ϕ?* This analysis computes the value of $Pr_M(\phi)$ that depends on the existence of a counterexample to $\neg\phi$, for the threshold θ. This computation is of polynomial complexity and, depending on the model M and the property ϕ, it may or may not terminate within a finite number of steps. In [26], a procedure based on the Chernoff-Hoeffding bound [28] was proposed, to compute a value for p', such that $|p' - p| < \delta$ with confidence $1 - \alpha$, where δ denotes the precision.

The SMC of BIP models is automated by the SMC-BIP tool (Figure 3) that supports both types of PBLTL properties. The tool accepts as inputs the PBLTL property, a model in SBIP and the confidence parameters α, β and δ. The SBIP model is then triggered iteratively by the SMC Core, a module that implements statistical testing algorithms [17], to generate independent execution traces. The execution traces are monitored in order to produce local verdicts. This procedure is repeated until an overall decision can be taken. At the end, the tool provides a verdict in the form of the probability for the property to hold true. The number of needed execution traces depends on the confidence parameters that are given as inputs. However, since the approach is designed for the validation of bounded LTL properties, it is guaranteed to terminate in finite time.

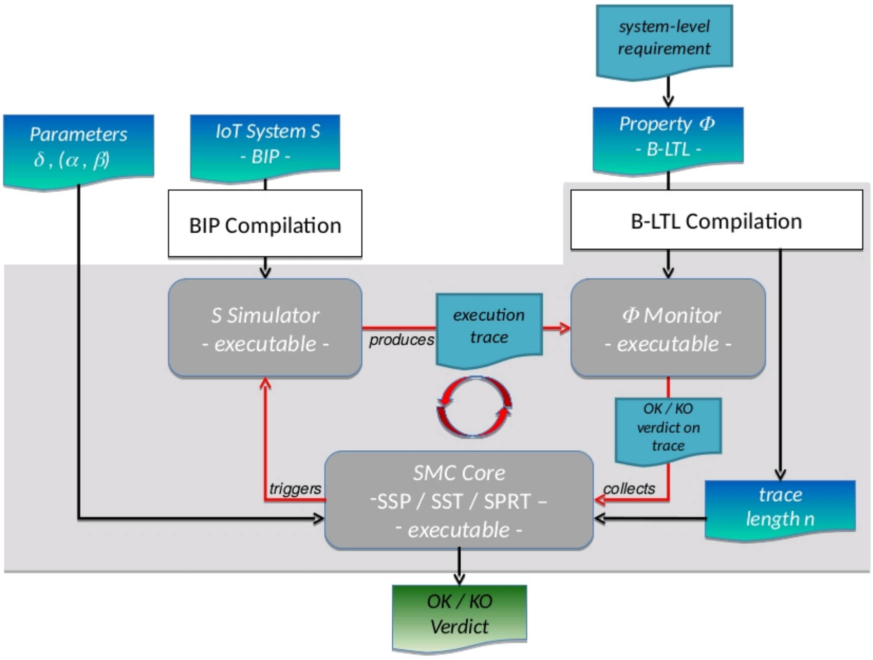


Figure 3: Architecture of the SMC-BIP tool for statistical model checking (source: SMC-BIP website[1])

3 THE BIP MODEL-BASED DESIGN FLOW FOR IOT SYSTEMS

An IoT project always starts from a set of requirements that can be classified in two major categories: (i) the functional requirements that are related to the application functionality, and (ii) the non-functional requirements related to the performance and the efficiency of the IoT system, along with its robustness characteristics. Functional requirements can be captured by formally specified safety and liveness properties, but we may be interested also for a quantitative characterisation of the correct operation of system under statistical assumptions for its external stimuli. Of particular importance are the non-functional requirements, such as the enforcement of bounded latencies due to e.g. packet collisions, and limited energy consumption.

We advocate a design flow using models that capture both the functional, as well as the non-functional aspects of behaviour across all abstraction layers of Figure 1, while supporting the separation of concerns during the system's design [29]. The separation of concerns is two-fold and involves not only the separation of application from the lower abstraction layers, but also the separation of

[1] http://www-verimag.imag.fr/Statistical-Model-Checking.html

computation from the communication. The former is of vital importance as it enables the application development independently from the IoT system architecture. The latter refers to the mechanisms and the primitives of the protocols employed in the network stack, which can be handled independently from the data processing. In this context, developers can model and build artifacts separately for the software and the various node architecture layers, which can be also reused in similar applications. Those reusable model artifacts are easily instantiated and parameterized, with respect to the particular system under design.

The design flow aims to the progressive development of the system, starting from the modelling and the implementation of the application functions up to their deployment onto the IoT system. The design philosophy is incremental in nature, since it is based on the hierarchical composition of simpler model artifacts, i.e. components, to form more complex *components.* An immediate consequence is that the debugging and identification of design errors in simpler components is easier and less time-consuming. In the course of the design flow, the developers should be able to find the optimal deployment - from the performance perspective at a given system scale - for the applications, while ensuring their proper functioning.

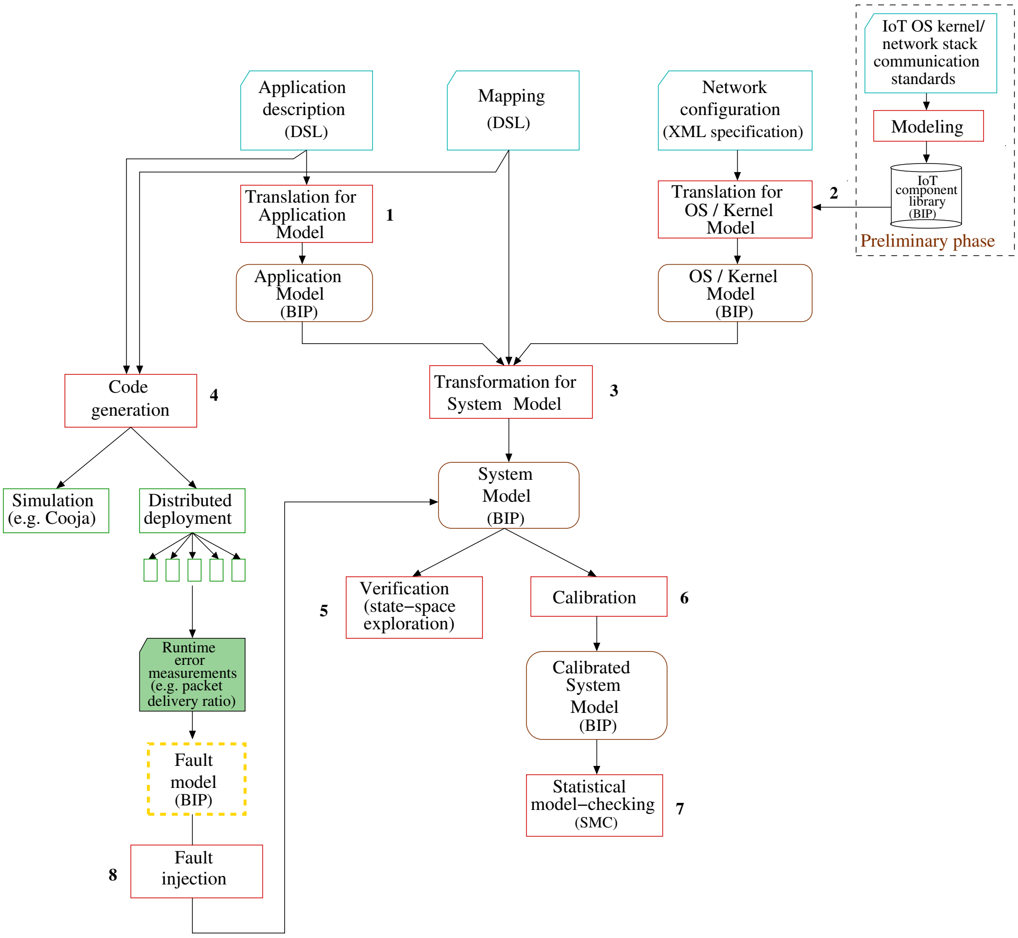
Our models rely on the BIP component framework. We take care of preserving the consistency between the BIP models and the corresponding application code by generating both from a single design definition written in a proprietary DSL. The DSL refers specifically to REST applications to be deployed onto IoT WPAN systems with Contiki OS nodes. The language provides XML-based constructs for the communication, the control flow and the scheduling of events in Contiki OS nodes, as well as for mapping the application's modules onto the system's nodes. The network configuration parameters are defined in XML files of the format proposed in [30]. Each model modification for fulfilling a violated requirement is respectively applied by the developer to the DSL design definition, from which the updated Contiki code is generated.

The design flow is based on analysis techniques for guaranteeing the qualitative and quantitative properties that capture the functional and non-functional requirements. The qualitative properties for safety and bounded liveness are formalised as observer automata [31] for monitoring the state of the BIP model and are then verified with the BIP tools for state space exploration. The quantitative properties are validated with statistical model checking [17].

Figure 4 provides an overview of our model-based design flow. We assume the availability of a library with model fragments for the applicable OS and SW/HW network stack, which can be reused in new IoT projects. All the necessary system components are instantiated from this library, according to the DSL design definition, which includes the IoT application's mapping onto the system's nodes. From these artifacts, it is possible to generate the BIP model of the IoT system. The overall approach consists of the following steps:

1. **Translation for the construction of the Application Model.** The design definition for a REST application in the DSL is translated into BIP. The structure of the DSL description is preserved and this allows to trace the analysis findings back to the design definition.
2. **Translation for the synthesis of the OS/kernel Model**. The BIP model fragments for the OS and the network stack (part of the network stack may be implemented by the HW architecture, as shown in Figure 1) are instantiated from the OS kernel library, through the translation of the XML-based network configuration file that determines also the components' parameterization and their interconnection.
3. **Transformation for the construction of the System Model**. The DSL mapping onto the system's nodes is manually edited[2]. It is used for transforming the Application and OS/Kernel models into a System Model. Appropriate BIP glueing code is generated from the defined mapping to connect the OS kernel model with the application model.
4. **Code generation.** The design and mapping definitions in DSL are used for generating code. The code can be either node-specific, i.e. ready to be deployed onto the distributed IoT system, or node-agnostic, if there is need for further validation within the OS simulator (Cooja). In any case, the code is deployed/simulated according to the specified application mapping onto the system's nodes.
5. **State-space exploration.** The verification of qualitative properties derived from functional requirements takes place by state-space exploration with the BIP tools. In case of a property violation, the original DSL definition (and the auto-generated BIP model) has to be repaired.
6. **Calibration.** The runtime characterization of the IoT application takes place by executing the generated code on the nodes and within the system environment. All influential hardware/

[2] The computation of the optimal solution would have to be based on a parametric mapping definition



IoT OS kernel/ network stack communication standards
Modeling
IoT component library (BIP)
Preliminary phase
Application description (DSL)
Mapping (DSL)
Network configuration (XML specification)
Translation for Application Model
1
Translation for OS / Kernel Model
2
Application Model (BIP)
OS / Kernel Model (BIP)
Code generation
4
Transformation for System Model
3
Simulation (e.g. Cooja)
Distributed deployment
System Model (BIP)
Verification (state−space exploration)
5
Calibration
6
Runtime error measurements (e.g. packet delivery ratio)
Calibrated System Model (BIP)
Fault model (BIP)
Statistical model−checking (SMC)
7
8
Fault injection

Figure 4: The BIP model-based design flow for IoT WPAN systems (numbered rectagles show the process steps)

software constraints are identified and subsequently added to the BIP System Model.

7. **Statistical model checking.** The validation of quantitative properties derived from functional and non-functional requirements takes place by SMC. If a property is not satisfied at the required level of confidence, the DSL definition has to be repaired.
8. **Fault injection.** The robustness of the BIP System Model is analyzed by considering various error behaviours, such as loss of bandwidth and radio interference as additive noise. The former increases packet losses and out-of-order deliveries and the latter causes error-prone access to the wireless medium and increases packet collisions [32]. For this analysis, additional components with the error behaviour are added, and further parameterization through proper runtime error measurements is required [20].

The design flow is iterated upon any change in the application's description or in its mapping, which follows the previous verification and validation step. The Step 8 takes place only when there are specific robustness requirements for the system under design.

In any case, the step 4 precedes the verification and validation of requirements, in order to enable the calibration of the system's model in step 6. This process aims to augment the model with timing information for computations and communications from executions of the actual system. The timing information for computations is derived from executing an application process on a system node. Communication times are induced by functions that use communication resources. In general, these times depend on the system's interactions with the environment, and on various interference factors that are not known in advance. In these cases, we can apply a probability distribution fitting technique, whereas the functional BIP model is transformed into a stochastic BIP model through the introduction of probabilistic variables that represent stochastic time evolution [17]. When the software runs to the completion of events and if it is possible to justify the absence of any interference, the measured execution times can be approximated by fixed times that are incorporated into the BIP System model.

3.1 Domain Specific Language for Contiki REST applications

The DSL for our model-based flow allows specifying a REST Contiki application in a single design definition file. This file is used as input to auto-generate both the BIP model and the C code to be deployed on Contiki nodes; in essence, it ensures the consistency between the BIP model and the application code across the design flow of Figure 4.

The language constructs used in REST Contiki applications are encoded into XML elements, and include essential control flow constructs, as well as actions for node communication and event scheduling. All these affect the validity of important functional and non-functional requirements for resource-constrained applications. Each element corresponds to a BIP code template and a template of Contiki C code. The BIP model and the C code are generated through a structure-preserving translation. Thus, every single part of the application model (port, component, data) can be traced to an XML element.

A REST Contiki application is represented by a <RESTapp> element (Listing 1) enclosing <module> elements (+ denotes one or more elements). A module contains client and server processes that will be loaded in one Contiki node and is identified by an *id*.

```
<RESTapp>
  <module>+
    (<client> | <server> )+
  </module>
</RESTapp>
```

Listing 1: DSL syntax for a REST Contiki application

A client process is encoded as shown in Listing 2 and includes optional *poll* and *exit* handlers (? = 0 or 1 elements), a block of initial actions (*init*) and a repeating *body* of actions. The body includes *wait* actions that enclose event handlers (*onEv* elements) for specific event types. A *wait* action causes the process to block until an event is received. If an incoming event is an EXIT or POLL or if it matches one of the enclosed event types, the corresponding handler is invoked.

Listing 3 shows the template that generates the Contiki code for a client process. The template contains placeholders for the code of the process's main elements (e.g. *exit*, *poll* and *body*). The *wait* is represented with a `PROCESS_YIELD` blocking statement and alternative `if` branches for the enclosed event handlers.

```
<client>+
  <poll> action+ </poll>?
  <exit> action+ </exit>?
  <init> action+ </init>
  <body>
    ( action*
      <wait>
        <onEv evType="MSG">*
          action*
        </onEv>
      </wait>
      action* )+
  </body>
</client>
```

Listing 2: DSL syntax for client

```
PROCESS_THREAD() {
 PROCESS_EXITHANDLER(/* exit */)
 PROCESS_POLLHANDLER(/* poll */)
 PROCESS_BEGIN();
 . . . . . . /* init */
 while (true) { /* body */
        . . .
        PROCESS_YIELD();
        if (ev == MSG){
                . . .
        }else if(...){ .... }
        . . .
 }
 PROCESS_END(); }
```

Listing 3: Contiki code template for client

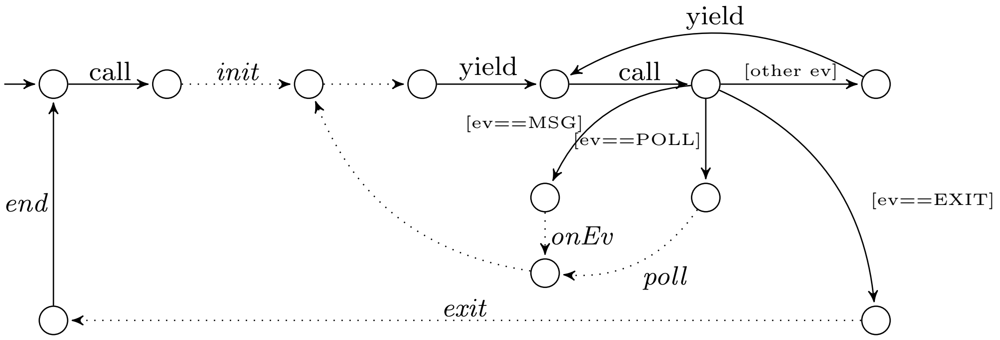


Figure 5: BIP template for the client (all event handlers of Listing 2 are composed in a single automaton)

The BIP component for the client is generated using the template in Figure 5. After the component starts (*called*), it performs the *init* actions and proceeds with the body loop. The *wait* is represented by: a) a *yield* port leading to a waiting state, b) a *call* port receiving an event, c) internal transitions (executed atomically with their preceding transition) that begin each event handler, d) an internal transition that returns to the waiting state, if the event doesn't match a handler. When the handler is finished, the component performs actions that follow the *wait* and eventually repeats the body. If an exit handler was activated, the component exits the body (*end* port), after the handler has finished.

More details for the DSL syntax are provided in Appendix A. Additionally, a complete design of an IoT system includes the mapping of application modules to the system nodes. The DSL syntax for defining this mapping is shown in Listing 4; this information is used for generating the BIP system model (process step 3 of Figure 4).

Each node's network configuration (*network-config*) determines a set of parameters, which are defined in our XML-based specification [30] (an example file is given in [29]), with default values that may be overwritten in auto-generated Contiki header files (e.g. `uipopt.h`). The same values are also used for the parameterization of the node's network stack model in BIP. For example, one such parameter is the maximum backoff exponent, which influences the network's waiting time, before another attempt to occupy the channel after a collision occurrence. The complete list of configurable parameters is provided in Appendix C.

```
<architecture>
    <node ip="string" module="QName">+
      network-config?
    </node>
</architecture>
```

Listing 4: DSL syntax for application deployment

3.2 BIP models for Contiki WPAN systems

Figure 6 outlines the BIP model structure for Contiki WPAN systems. In overall, the System model comprises two layers, the REST Application Model (AppModel) and the Contiki Kernel (ConKernel). The Application Model consists of BIP components for the REST modules that define the client-server interactions and their constraints, as they are derived from step 1 of Figure 4. Based on the REST modules mapping to Contiki nodes, the System Model integrates the application model with the BIP components from the OS kernel library and the network stack, for communication through the channel. For the lower level hierarchy we use phrase structure rules, where each rule refers to a BIP compound (shown in "<" and ">") in the left and its constituents (enclosed components) in the right part:

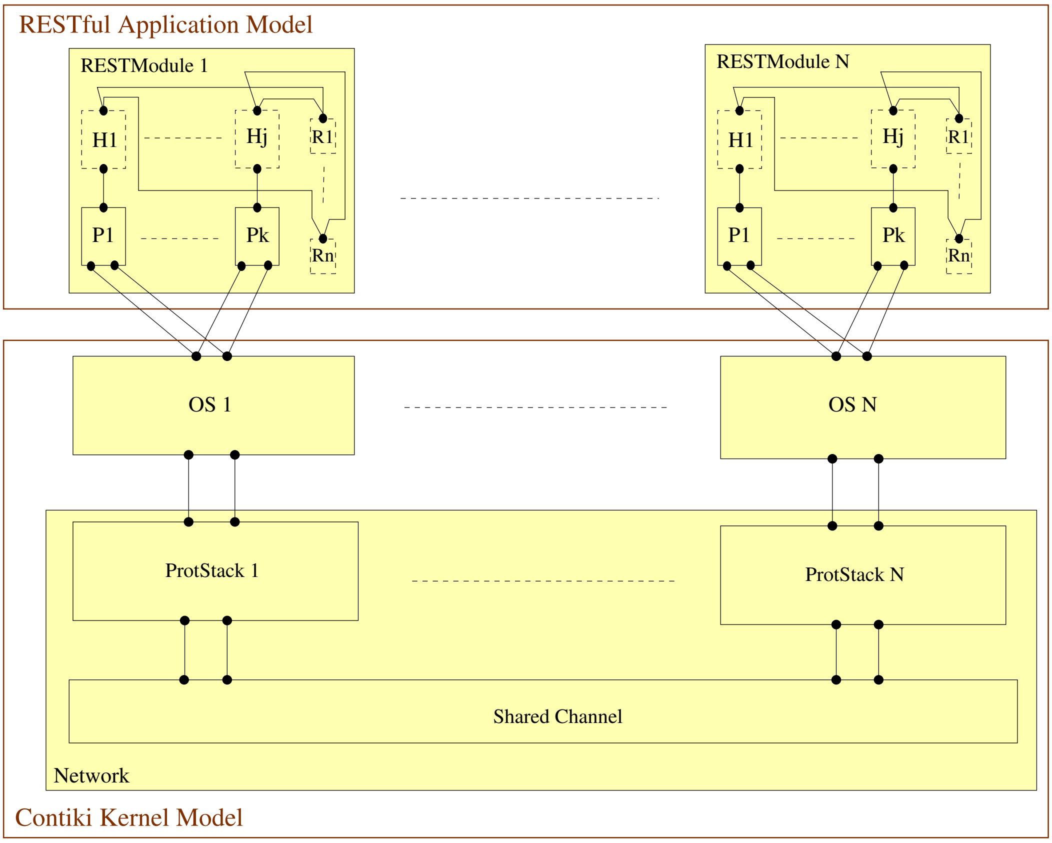


Figure 6: BIP model structure for Contiki WPAN systems

$\langle SystemModel \rangle$ ::= $\langle AppModel \rangle$ $\langle ConKernel \rangle$

$\langle AppModel \rangle$ ::= $\langle RestModule \rangle$+

$\langle RestModule \rangle$ ::= Process+ (Resource ResHandler+)*

$\langle ConKernel \rangle$ ::= $\langle OS \rangle$+ $\langle Network \rangle$

$\langle OS \rangle$ ::= Scheduler Timer CommHandler

$\langle Network \rangle$ ::= $\langle ProtStack \rangle$+ Channel

$\langle ProtStack \rangle$ ::= MsgSender MsgReceiver

The <AppModel> consists of several <RestModule>, each of them with a number of processes and (optionally) REST resources with their handlers. <ConKernel> includes <OS> components for each node and the <Network>, encompassing components for the network stack and the communication medium (channel). The interactions between <RestModule> and <OS> are detailed in Appendix B. <OS> models the scheduling of interprocess and remote communication (includes the Scheduler, the Timer and the CommHandler [29]). The granularity of behaviour in components ensures that all interleavings of events in a DSL definition are taken into account.

The Scheduler maintains a FIFO queue with the posted asynchronous events, boolean flags with poll requests and a call stack with the active processes, i.e. those that were called and subsequently paused after having posted a synchronous event. When a synchronous event has been handled, the call stack is used to transfer the control to the right active process. A cycle is initiated periodically (period $p_{scheduler}$), in which the Scheduler first sends the requested poll events and then dispatches an asynchronous event from the FIFO. The cycle is completed, when the call stack is emptied. If the queue is full, an error is returned to the process.

The Timer receives timing requests from <RESTModule> processes and stores them in a stack. All requests have a mode, which allows setting, resetting, restarting and stopping a timer. The time advancing is modeled through an interaction that synchronizes all model components. The granularity of a time step affects the simulation efficiency and analysis scalability, and it is therefore determined by a configurable parameter. The remaining time for the next timing interaction is computed separately for each component, and the time advances directly by the minimum number of time steps amongst all components [29].

The CommHandler models the TCP/IP processing and interacts with <Network> (packets are stored in a transmission buffer *TxBuffer* or a reception buffer *RxBuffer*). <ProtStack> includes the *MsgSender* (Figure 2) and *MsgReceiver* components, for message transmission (resp. reception) with the CoAP or HTTP protocol. Channel (Figure 2) implements behaviour for message transmission and for resolving collisions in simultaneous transmission requests.

The network stack's performance is modelled by proper parameterization of the <Network>. We have the fixed and the modifiable parameters (with default values), shown in Appendix C.

3.3 Calibration

The System Model omits characteristics, which can be measured only during the code execution on the nodes and within the system environment. This information includes the characterization of sensor data and HW/SW performance factors, like the time or other resource costs to be quantified at runtime. In the former case, we need to derive probabilistic distributions that fit to measurements of sensor data. In the latter case, it is necessary to employ performance characterization methods, such as the *process profiling* technique [29] that allows measuring isolated blocks of code. The values obtained from this analysis are injected as parameters to the System Model and we eventually obtain the Calibrated System Model (Step 6 of Figure 4).

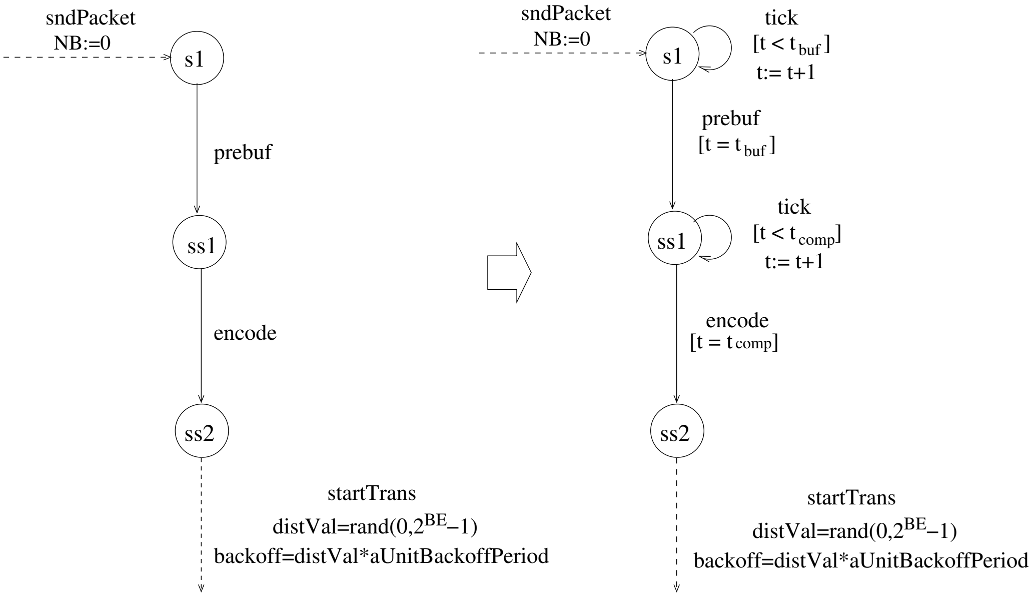


Figure 7: New transitions/guards for the calibration of the ProtStack.MsgSender component of Figure 2

Let us consider the time for pre- and post-buffering of each message transmission, which can only be measured during the code execution. The Contiki code for an IoT application is generated first (Step 4 of Figure 4) and then instrumented with *clock_time* calls. For the pre/post-buffering phase, the `memcpy` function is isolated between two subsequent calls of *clock_time*. The elapsed time is computed by subtracting the two recorded time instants.

Figure 7 depicts the calibration of the *ProtStack.MsgSender* component. We focus on the $s0 \rightarrow s1$ and $s1 \rightarrow s2$ transitions of Figure 2, labelled by the *sndPacket* and *startTrans* ports. Two internal transitions (not involved in interactions) are inserted between *sndPacket* and *startTrans*, called *prebuf* and *encode*. A model transformation is finally applied: two loop transitions are introduced, as shown on the right-hand side of Figure 7, which are annotated with discretized time durations (i.e. t_{buf} and t_{comp}) and guards that block $prebuf$ and $encode$ until the time delays have been elapsed.

3.4 State-space exploration

For the state-space exploration, each property is formalised as an observer automaton [31] monitoring the events in the BIP model that are relevant to the property, i.e. a set of interactions. Every such event triggers a state transition, which may cause the observer to reach an error control location with no outgoing transitions. If the error location is reached, the property is violated. Observers are attached to the model using broadcast connectors, which allow the model's components to trigger the observers by excluding the other way round. Thus, observers do not interefere with the model and our analysis is sound. To avoid state explosion, it suffices the exploration to be limited to execution scenarios that generate all relevant event interleavings for the examined properties.

As an example, Figure 8 shows the observer automaton for the requirement: *The client never sends redundant messages to a server.* The error is reached from $s2$ and $s3$. In $s2$, the client has sent a request (*Cli_sndMsg* port) that has been acknowledged (*NetwCli_recvAck* port), while the server has not yet transmitted (*NetwSrv_transmit* port) or aborted the transmission (*NetwSrv_endSnd* port) of response. In $s3$, the response has been transmitted to the client, but it has not yet been received (*Cli_getMsg* port). Any request sent while in one of these two locations is considered redundant.

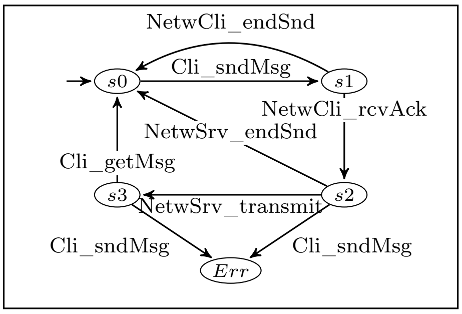


Figure 8: Example observer automaton for the formal verification of a qualitative property (no redundant service requests)

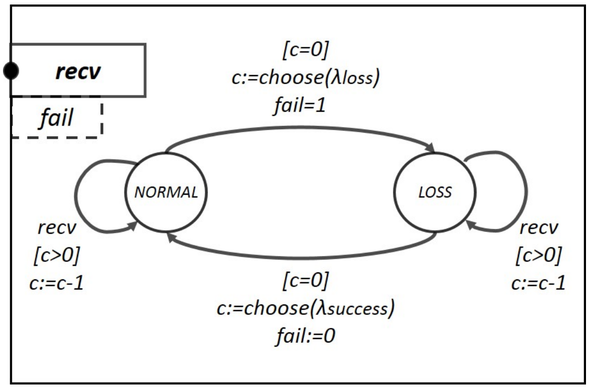


Figure 9: Example FaultHandler automaton - packet is ignored if fail = 1

3.5 Fault injection

The fault injection (Step 8 of Figure 4) is used when there are requirements for the robustness of the IoT system. One such requirement is the tolerance of extensive bandwidth loss, which assumes fault behaviour for studying the impact of consecutive long packet delays or packet losses.

As an example, Figure 9 depicts a FaultHandler component that receives the transmitted packets (*recv* port) and decides whether they will be delivered to their destination. This choice is handled through the *NORMAL* and *LOSS* locations, which represent the successful transmission and the case of delayed or lost packets. The FaultHandler remains in each location, for as long as the number of consecutive transmissions is positive. This number is chosen from two probabilistic distributions, $\lambda_{success}$ and λ_{loss}. The FaultHandler is added to the Calibrated System model, which is then validated using SMC.

4 CASE STUDY

4.1 General description

The design of a building automation SCC application with digital and analog sensors is now illustrated using the BIP model-based design flow of Figure 4. The application controls the temperature and detects motion in offices using passive infrared (PIR) sensors. A ZIG001 Temperature-Humidity sensor is installed in each office, along with the low power MS-320LP PIR, both from Zolertia[3]. A zone-controller (acting as Client) reads temperature measurements (shown as t in Figure 10) and turns on a thermostat, if the temperature exceeds the desired level ($t > ub \vee t < lb$). During non-office hours, the controller runs into the energy-saving mode, and the desired temperature is reduced. If there is motion detected (Figure 11) during non-office hours, an intrusion alarm is activated. The intrusion alarm notifies the subscribed devices inside or outside the building (e.g. smartphones). If there is motion during office-hours, the lights switch on.

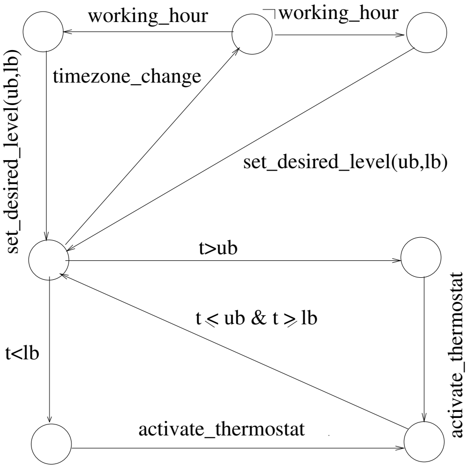


Figure 10: State flow for temperature control by the zone-controller (temperature t should be in $[lb, ub]$)

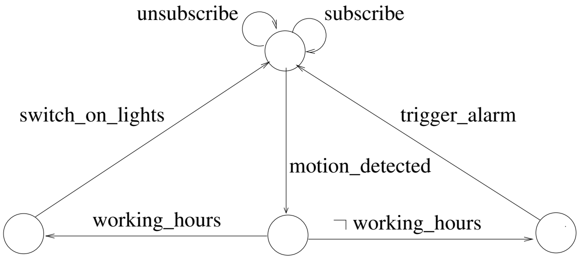


Figure 11: State flow for motion detection by the zone-controller (alarm triggered in non-office hours)

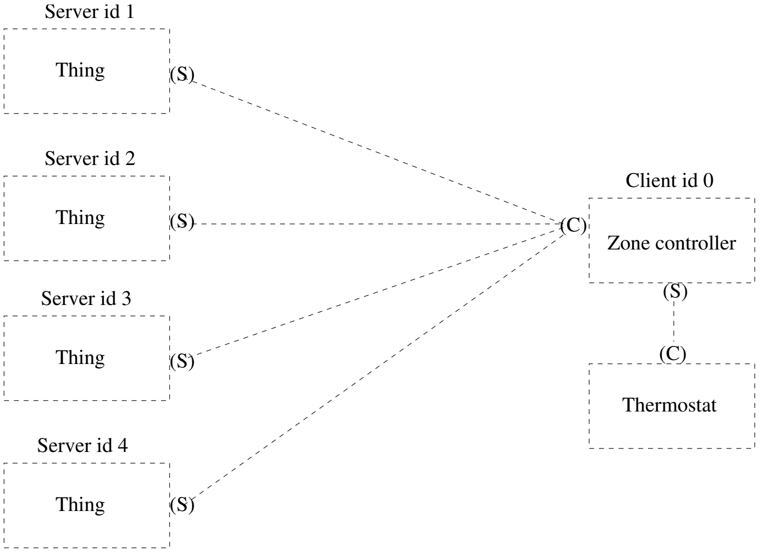


Figure 12: Node topology with clients and servers in the building automation system

The system consists of 5 nodes with the zone-controller acting as client of 4 REST servers using CoAP (Figure 12). Each server owns a temperature resource for a ZIG001 and a motion resource for a MS-320LP PIR. The client runs two Contiki processes, one for sending unicast GET requests to the servers and one for periodically observing their motion resource. The GET requests are sent with a fixed transmission period of 1s. The observation of a resource begins with a registration request and

[3] http://wiki.zolertia.com/wiki/index.php/Z1_Sensors

is cancelled with a de-registration request. Upon the state change of a resource, the server sends a CoAP notification to the registered client, which acts according to the time of day. Message receipts should be acknowledged by the server. However, if the client does not receive a response within a deadline, the request is re-transmitted.

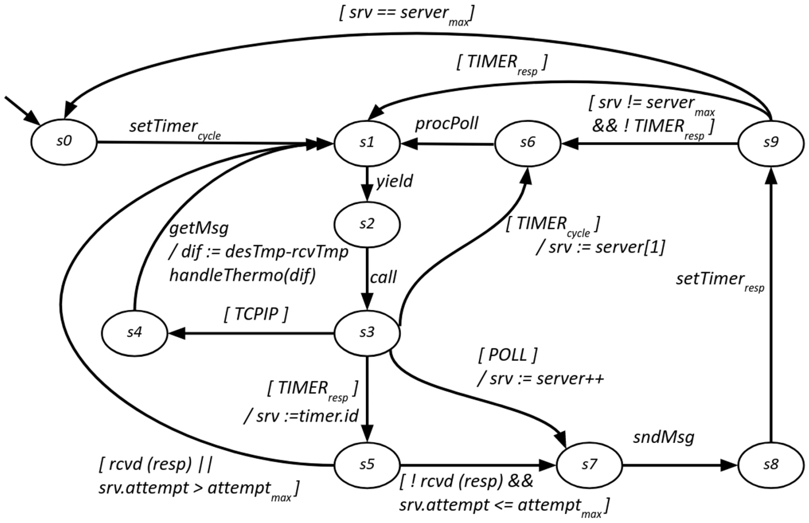


Figure 13: Client process of the building automation application

Figure 13 depicts the behaviour of the client process's body for receiving temperature measurements[4]. A cycle of measurements starts upon a $TIMER_{cycle}$ event (timer expires). The process then sends a request (unicast message) to a server, and asks to be polled for contacting another server, until all servers have been contacted. For each request, a deadline timer is set ($setTimer_{resp}$) for waiting the response. If the deadline expires and no response has been received, the client resends the request to the same server up to a maximum number of attempts. When the response of a server is received (TCPIP event) the temperature is read and if it differs more than two degrees from the desired level, the thermostat is turned on. The client starts a new cycle, when all servers have sent measurements or the maximum number of attempts has been reached.

4.2 Application of the BIP design flow

We focus now on the individual steps of the design flow in Figure 4, for the outlined SCC application.

Step 2: synthesis of the OS/kernel model

The case study was based on the detailed modeling of the Contiki OS by taking into account all kernel interactions with the application layer and the network. This allowed us to deliver the first OS kernel library [29].

Step 3: construction of the System Model

The generated BIP system model for the case study (4850 lines of code) consists of 30 atomic components for the AppModel and 26 components for the ConKernel; it includes 430 connectors and 805 transitions. The time step of the model was based on the transmission time per bit through the Contiki network stack. This is given as the inverse of the data rate of a Contiki network access point. The smallest transmitted data unit is one symbol (4 bits) and its transmission time is:

$$symbolPeriod = \frac{4}{dataRate} \tag{1}$$

For an access point to a wireless medium operating at the 2.4 GHz band, the data rate is 250 kbps and from (1) the symbol period is $16\mu s$. Thus, our timing abstraction ignores delays smaller than the inverse of this data rate, which is $4\mu s$. This adjustment allows for a much more fine-grained timing

[4] To ease readability, a simplified view of the behaviour is shown from that derived as in Figure 5. Guards are shortened (e.g. *POLL* instead of *ev==POLL*) and transitions not relevant to the application logic are omitted (e.g. the *EXIT* event handling, the *msgSnt* and *timerSet* transitions of the templates of Figure 17 that respectively follow *sndMsg* and *setTimer*.

model compared to the one of the Cooja simulator, which is in the *ms* scale. Every parameter of the model that is associated with a time delay can be quantified with an adequate degree of fidelity.

Step 5: state-space exploration

IoT applications are prone to event scheduling delays. Thus, clients set deadlines for expected responses before re-sending the requests. Each deadline is tuned, such that it is feasible to receive a response and avoid sending redundant requests. For the case study, the following functional requirements were formalised as qualitative properties that were checked by state-space exploration:

Functional Requirement FR1 The client reaches `PROCESS_END`.
FR2 The client never sends redundant messages to a server.
FR3 The client collects measurements from each sensor at least once in a specified period.

FR1 specifies a liveness property stating that the client shall terminate. The property is violated if the process does not exit by itself and neither receives an EXIT message from another process, i.e. the corresponding actions are omitted in the code or they are not reachable at runtime. For FR2, a deadline must be set, in which the client obtains a response and avoids sending redundant requests. Every such request is monitored by the observer automaton in Figure 8; the property may not be satisfied due to node communication and event scheduling delays that invalidate the set deadline. FR3 implies finding a period for sensing the environment, in which the client will have collected all necessary measurements.

The scenario considered for the state-space exploration involved a limited number of messages to be sent by the two client processes. Specifically, two GET requests, a registration and a de-registration request are sent to each server. Moreover, the GET requests could be attempted twice. The above scenario is sufficient for the generation of all the necessary interaction interleavings that are relevant to the verification of requirements FR1 to FR3.

Step 6: calibration

The calibration was performed based on the execution of the generated code. The temperature distribution was fitted using several measurements taken at random instants during the course of a day. For the motion detection, we fitted a normal distribution with mean $\mu = 1.5$ Volts[5] and standard deviation $\sigma = 1.5$ Volts (we used the distribution fitting method in [33]).

The System Model was augmented with timing information for computations and communications measured in executions of the system. Specifically, (i) the time for the packets' IP header compression/decompression was measured by linking the Contiki OS with the 6LoWPAN protocol implementation of the HC1/HC2 encoding mechanisms [34], whereas (ii) the time for pre- and post-buffering of each message transmission was measured by locating a message and copying its fields from/to the buffers[6]. These time costs are fixed, because the computations are applied to a fixed size input and they do not interfere with other computations until they finish.

Steps 7 and 8: statistical model checking and fault injection

A key functional requirement for an SCC application is to achieve its ultimate control objective:

FR4 Room temperature is maintained within [-2,2] C degrees difference from a user-defined level.

FR4 depends on the temperature variability of the system environment in combination with the client's ability to collect data sufficiently often to act on time (data collection is prone to delays). This requirement was formalised as a stochastic temporal property in PBLTL [26] and the property was validated with the SMC-BIP tool. Moreover, the following key non-functional requirements refer to the performance of the building automation system:

Non Functional Requirement NFR1 Rapid detection of movement during non-working hours, based on the PIR's voltage level.

[5] Motion detection is sensitive to the distance, which impacts the sensor's voltage level. Here we observe the current voltage level of the sensor and not its binary output. A threshold 0.5 V was used with the sensor generating a motion event when the current voltage level is higher than the mean value augmented by the threshold (1.5 + 0.5 = 2V).

[6] The model's time step was much larger than the time taken to store the messages in the transmission/reception buffers.

NFR2 Memory saving by properly sizing the message buffers used by the Contiki network stack in each node.

NFR3 Avoidance of overflows in the asynchronous event queue of each node.

NFR4 Relatively low collision rate in the channel, in order to avoid extensive latencies, which deteriorate the network performance and increase the probability of packet losses.

The mentioned non-functional requirements were formalised as stochastic temporal properties in PBLTL, which were accordingly validated on the calibrated BIP system model.

Regarding the fault injection (step 8 of Figure 4), we used the FaultHandler component in Figure 9 for studying the system tolerance to extensive bandwidth loss. The FaultHandler was parameterized using probabilistic distributions derived from the analysis of debugging traces [29] obtained from executing the code on the node topology of Figure 12.

4.3 Experiments and results

The state-space exploration took place within the Contiki 16.04 environment running on a workstation with an Intel i5-3230M CPU@2.60GHz (4 processors), 6GB RAM and 500GB HDD. The SMC experiments were conducted within the Instant Contiki 2.6 environment running in a virtual machine with one CPU core, 1GB RAM and 9GB hard drive.

FR1 was verified as a bounded liveness property using an observer automaton that monitors whether the client has reached `PROCESS_END` before the system model terminates. For FR2, we checked whether the property holds for the temperature monitoring process, using various deadlines (112ms, 160ms and 800ms). The requirement was satisfied for a 160ms deadline, if the motion detection process is not observing simultaneously and it is violated for a 112ms deadline. These results show that the it is less likely for the client to get responses within short deadlines, hence it sends more redundant requests. However, it is possible the client to get all responses even at short deadlines, provided that the motion detection process does not simultaneously consume network and node processing resources. The FR3 is satisfied for a period of 5.6s even when the motion detection is operating.

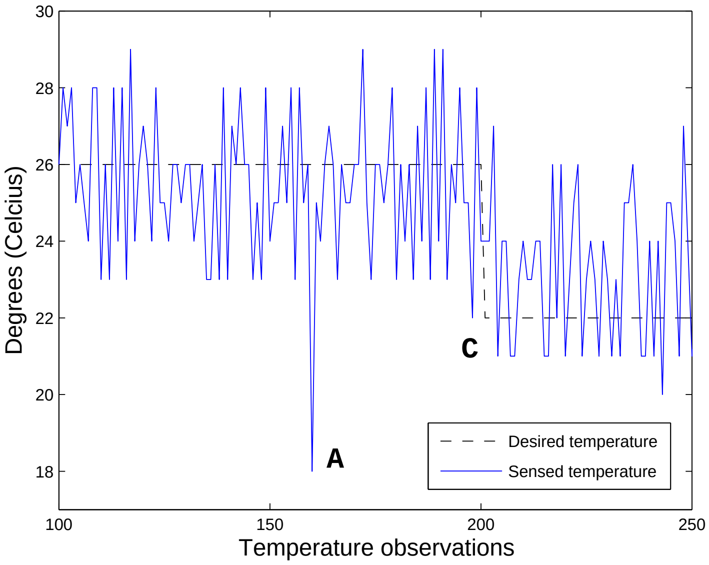


Figure 14: Temperature observations (in Celcius) for our building automation application

For FR4, we checked $\phi_1 = |RecvDegree - InputDegree| \leq 2$, where $RecvDegree$ is the temperature sensed by the ZIG001 sensors and $InputDegree$ is the desired temperature level. We found that $P(\phi_1) = 0.6$, due to the zone-controller responsiveness to temperature changes and fluctuations attributed to external factors. Figure 14 shows part of the obtained observations, with the temperature often exceeding the acceptable range, like in point A. Such a rapid change might be attributed to an abrupt change in the environment, as the opening of a window. In point C, the desired temperature has been changed, the zone-controller has perceived the change and the temperature was then reduced by the thermostat.

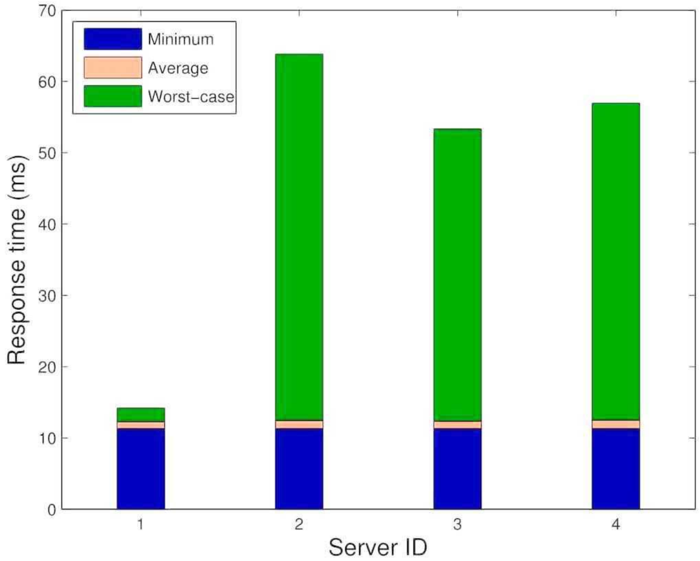


Figure 15: Transmission times (ms) for motion detection with faults injected in the Calibrated System Model

For the non-functional requirements NFR1 - NFR4, we analysed two sets of execution scenarios: the first set was obtained using the Calibrated System Model (step 7 of Figure 4) and the second set by including the FaultHandler component of Figure 9 (step 8 of Figure 4).

For the second set of execution scenarios, Figure 15 shows the transmission time of messages for all servers upon changes in the motion resource. These times are classified in three categories (shown in different colours), namely the minimum observed, the average and the worst-case time. We observe that the worst-case times are significantly different from the two other times, i.e. when there are no collisions or message delays, except from Server 1. This happens because Server 1 does not encounter additional transmission delays, since it is the first to which the zone-controller sends the room's temperature and there are no additional messages to transmit/receive, before the motion resource change messages.

The SMC experiments for NFR1 - NFR4 generated the following results:

NFR1 We checked the property $\phi_2 = T_{PIR} \leq T_{trans}$, where T_{PIR} is the worst-case message transmission time for the motion resource and T_{trans} the period for a regularly transmitted message, i.e. a client request for the temperature resource (1s). In the first set of experiments, T_{PIR} did not exceed 32 ms, hence $P(\phi_2) = 1$. In the second set with the presence of the FaultHandler, a higher number of packet collisions occurred with T_{PIR} being approximately 63ms (server ID=2 in Figure 15). Nevertheless, the property still holds, since T_{PIR} remains smaller than T_{trans}.

NFR2 We checked the property $\phi_3 = (size(RxBuffer) < B)$ in the first set of experiments, where B is a bound for the reception buffer size of the protocol stack[1]. B depends on $p_{scheduler}$ (cf. Section 3.2), which affects the rate in which the MsgReceiver removes messages from the buffer. When $p_{scheduler} = 10\mu s$, $P(\phi_3) = 1$ if $B = 2$. When $p_{scheduler} = 10ms$, B would have to be 10, so that $P(\phi_3) = 1$. Thus, $p_{scheduler}$ must be small enough to avoid adjusting the reception buffer size.

NFR3 We have analysed the property: $\phi_4 = (size(FIFO) < MAX)$, where $size(FIFO)$ represents the size of the Scheduler's FIFO queue and $MAX = 10$. This property holds true ($P(\phi_4) = 1$) in both sets of experiments.

NFR4 We analysed the property: $\phi_5 = (NC \leq 1)$, where NC is the number of re-transmissions following a collision. In the first set of experiments we had $P(\phi_5) = 0.75$, which implies a limited number of collisions. In the second set of experiments, we observed a significant collision rate that resulted in $P(\phi_5) = 0.5$.

[1] The bound on the reception buffer size can be adjusted by the parameter $MAX\_NUM\_QUEUED\_PACKETS$ of the Contiki kernel; the default value is 2.

5 DISCUSSION

5.1 Benefits of the BIP design flow

The design flow of Figure 4 supports the progressive development of IoT WPAN systems using BIP models that capture both functional and non-functional system aspects. BIP is component-based, which is essential for enhanced productivity through reuse of model artifacts. The building automation case study in Section 4 aimed to test the effectiveness of our approach. Through the invested effort, it was possible to deliver tools (DSL and code generator) and a component library with model fragments for the Contiki OS and the network stack. Table 1 reports figures for the required effort and the size of the model and code artifacts for the case study. Each row refers to a particular step of the design flow, but the effort for steps 2 and 4 includes development work, which was done once for the tools and the library of components but can be reused in similar projects.

Step of Figure 4	**Effort**	**Scope**	**Product**	**Lines of code**
1. App design definition	4 days	Application-specific	DSL	120
2. Network configuration	6 hours	Reusable	XML	70
2. IoT comp. lib. & *ConKernel*	7 weeks	Reusable	BIP models	2530
3. App mapping & transform.	4 hours	Application-specific	DSL	40
4. Code generation	8 weeks	Reusable	C	1168
6. Calibration	4 days	Reusable	BIP model	70
8. Fault injection	3 days	Reusable	BIP model	70

Table 1: Effort and design artifacts for the building automation case study

Model	**Components**	**Connectors**	**Transitions**	**Lines of code**
Application model	30	130	223	856
OS/kernel model (*ConKernel*)	26	300	582	3924
System model	56	430	805	4780
Calibrated system model	56	430	820	4850
Fault model	5	13	25	70

Table 2: BIP model statistics for the building automation case study

Step of Figure 4	**Verified properties**	**Satisfied properties (without faults)**	**Satisfied properties (with faults)**	**CPU time**	**Memory**
State-space exploration	Deadlock-freedom	✓	N/A	5h 13 min	4500MB
	FR1	✓			
	FR2	✓			
	FR3	✓			
Statistical model checking	FR4	60%	55%	2h 10 min	956MB
	NFR1	100%	100%	1h 35 min	720MB
	NFR2	100%	100%	1h 21 min	630MB
	NFR3	100%	100%	0h 24 min	780MB
	NFR4	75%	50%	5h 4 min	918MB

Table 3: State-space exploration and statistical model checking statistics for the building automation model

Table 2 reports statistics on the complexity of the generated models. A substantial difference appears between the complexities of the models for the Application and the OS/kernel (*ConKernel*). This is due to the detailed modeling of the Contiki OS and network stack functionalities in the library of model components. We thus ensure that all relevant events for the analyzed properties are taken into account, whereas the models should be also valid for additional properties that may be of interest in other Contiki WPAN systems.

Table 3 presents statistics for the used resources in steps 5 and 7 of Figure 4, i.e. the state-space exploration and the SMC of the building automation model. The verification of deadlock-freedom and the properties for FR1 - FR3 consumed 4.5GB of RAM and was completed in 5h 13 min. The

CPU time for the SMC of the properties for FR4 and NFR1 - NFR4 varies depending on the used confidence and precision parameters (cf. Section 2.3.2).

A comparison of BIP's SMC efficiency with the Cooja simulation is certainly interesting (recall that the granularity of our model's time step is in the order of a few μs, whereas Cooja's time step is in the order of ms). For this purpose, we conducted simulations of the building automation system during office hours, as well as when it is operating over the whole day including office and non-office hours. The results are shown in Table 4.

Simulated time	**Cooja simulation**	**BIP SMC**
8h	1h 03min	1h 20 min
24h	3h 27min	3h 02min

Table 4: Simulation time for the building automation model

Here, it is important to recall that Cooja works with a fixed time step advance of the simulation clock, while the BIP simulation advances the clock directly to the time of next event, for all system nodes (cf. Section 3.2). The reported CPU times for BIP SMC include as many experiment replications as necessary for providing an accurate verdict.

From the system design point of view, the benefits of our approach stem from the motivating principles in Section 3; an informative comparison with other design methods is exposed below.

5.2 Limitations

The design flow limitations in its present form concern with its applicability scope, the expectations for experience of its users on formal methods, the degree of process automation and some scalability issues related to the model complexity, the size of systems under design and the supported analyses.

The current applicability scope is the Contiki-based WPAN systems, due to the available model fragments in our IoT component library. For widening the application to other Contiki system architectures, it is essential to enrich the component library, as well as to customize the specification file for the network configuration and the translation for the synthesis of the OS/kernel model (step 2 of Figure 4). If the focus is extended to other types of applications than Contiki REST, or to other IoT operating systems, then there is need to modify or even create a new DSL. This implies adaptations or respectively new implementations for the translator used in step 1, the model transformation in step 3 and the code generator in step 4 of Figure 4.

If the analysis is restricted to the requirements of our building automation system, there is no need for formal method experience by the designers. For additional requirements concerning timing, memory or thermal aspects, there is still need for some basic knowledge on automata, in order to formulate the new requirements in terms of the BIP system model. For requirements focusing on other aspects, say for example the energy consumption, we need to add appropriate power models for quantifying the energy resources used in computations and communications during the execution of system model. Related work in this direction has been presented in [35].

The degree of process automation is, as always, a matter of balancing usability and flexibility concerns in the provided tool support. We opted to limit the automation to steps 1, 2, 3 and 4 of Figure 4, while retaining the ability to manually edit the BIP model during steps 5, 6, 7 and 8, in order to have more space for experimentation during our analyses.

Finally, for the scalability to larger IoT systems, it is important to note that our analyses took place for a limited number of nodes. This is due to the detailed representation of interactions within the *ConKernel* model, since it was generated from our component library in step 2 of Figure 4. Such detailed modeling is essential for the state-space exploration and for calibrating the model with sufficient accuracy; it allows validating very diverse requirements and opens prospects for additional requirements in future works. Moreover, for IoT systems of a larger size there is no need for a BIP model that will include all of the system's nodes; for the state-space exploration we only need a model with representative instances of those types of nodes, which suffice to generate all relevant interleavings of events for the properties. For far more complex IoT applications and possible SMC scalability issues, the BIP model can be abstracted using automated stochastic abstraction techniques [36] for identifying those events that are significant for the properties of interest.

5.3 Comparison with competitive design methods

The design of IoT systems involves rapidly evolving technologies, diverse applications and a fragmented landscape of operating systems/middleware with various development tools. The major incentives for a model-based design process are: (i) the overall design complexity, due to the high heterogeneity of interconnected things and their interaction modes, and (ii) especially for IoT WPAN systems, the limited computational and energy resources. System models allow reasoning about certain properties of the system's behaviour, and serve as a specification that will lead to a physical implementation of the system, which is compliant with the model. They also provide a means for exploration of different design alternatives. The Flex-eWare model [37] is a general purpose component model for distributed embedded systems that aims to unify model driven and component-based software engineering across many different application domains. This is achieved by integrating generic elements in its metamodel that are instantiated by model libraries. All other design methods in this comparison focus on specific IoT system architectures or application domains.

Table 5 summarizes the main characteristics of the design/analysis methods found in related bibliography and compares them with the BIP design flow of Figure 4. Most approaches are model-based, but only the BIP flow and [38, 39, 40, 41, 42, 43] are grounded on languages with formal semantics.

The validation of quantitative specifications or the robustness analysis of systems is supported by some methods, but only [42] and the BIP flow support the verification of qualitative properties, with the latter covering all categories. Moreover, most methods do not cover the entire design cycle (including code generation and deployment) with the notable exceptions of [38], [40], [42], [43] and the BIP design flow. Some methods are limited to the analysis of WSN systems.

In terms of the tools that support the various forms of analysis, most of the approaches in Table 5 provide or utilize various simulation frameworks with the notable exception of the BIP flow along with [39], [40], [42] and [43], which support statistical model checking and/or other formal analyses. Finally, a noteworthy number of the shown methods aim to the generation of code for REST services running on the Contiki OS or other operating system.

6 CONCLUSION

We presented a model-based design flow for resource-constrained IoT applications using the BIP component framework. Currently, the provided support concerns with the design of REST service-based applications for WPAN systems with nodes running the Contiki OS. The application-level model and the IoT system model in BIP are generated from a design definition of the REST application and its mapping to Contiki nodes that are both expressed in a domain specific language. This design definition is also used for auto-generating the code to be deployed to the nodes, thus preserving the properties of the validated models. All tools and models of the design flow of Figure 4 along with technical instructions on their use are available online [7].

The described approach was applied to a building automation application running on one client and four REST server nodes with various forms of interactions between the connected things (periodic transmission of temperature measurements and event-driven transmission of motion detection signals). We verified functional requirements related to service responsiveness and subsequently we validated the system's operation under statistical assumptions for its external stimuli. The analysis also included important non-functional requirements for WPAN applications and their robustness with respect to a fault behaviour reprsenting an extensive bandwidth loss.

As the key features of the overall approach, we highlight the following:

- BIP promotes a component-based design philosophy that allows locating design errors to specific components and supports the modular design and reuse of model artifacts.
- The design flow supports the separation of concerns between the application design from the lower-level system aspects, and the separation of computation from the communication. A direct consequence is that developers can model and build artifacts separately for the software and the lower node architecture layers.
- Both functional and non-functional system aspects are captured in the BIP models that enable a wide range of analyses using the BIP tools and the SMC-BIP.

As future work, it is worth to consider the extension of our domain specific language towards supporting MQTT (Message Queue Telemetry Transport) clients and servers, as in [45]. To this respect, a publish-subscribe model has to be integrated, such as the BIP model described in [50]. An interesting direction is the detection of compute-bound event loops [51] and the analysis of their robustness, which is defined as conjunction of two correctness properties [52], event serializability and event determinism (executions within each event are insensitive to the interleavings between

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

Table 5: Logical comparison of design-time analyses and design methods for IoT systems

Design method /analysis	**Focus**	**Scope**	**Qualitative properties (verification)**	**Quantitative properties (validation)**	**Robustness analysis**	**DSL & tool support**
[21]	cloud-centric IoT	app design	✗	✗	✗	Aneka Platform-as-a-Service
[38]	REST \Contiki or TinyOS systems	model-based, app design, sys. design, code gener., deployment	✗	✗	✗	graphical Matlab tools, network/hw-in-the-loop simulation
[44]	Sense\Comp. \Control apps	model-based, app design, code gener., deployment	✗	✗	✗	DSL, progr. framework generator, runtime, scenario simulator
[45]	Android & JavaSE devices with MQTT	model-based, app design, sys. design, code gener., deployment	✗	✗	✗	DSL, compiler, linker, runtime system
[46]	iSense OS for WSNs	code gener.	✗	✗	✗	DSL
REMORA [47]	services for WSNs (inc. REST)	app design	✗	✗	✗	programming language
WSN-DPCM [48]	services for WSNs (inc. REST)	model-based, app design, sys. design, code gener., deployment	✗	✗	✗	middleware, network simulation
[49]	services for WSNs (inc. REST)	model-based, app design	✗	✗	✗	DSL
[39]	WSN analysis	model-based, performance \dependability	✗	✓	✓	WSN topolo- gy (GUI), trace analysis
[40]	Arduino,Raspberry Pi (POSIX), Robot OS	model-based, app design, sys. design, code gener.	✗	✓	✗	ThingML (DSL), stat. model checking
[41]	WSN analysis	model-based, performance & dependability	✗	✗	✓	DSL, analytical & behavioural simulation
BeC^3 [42]	REST Contiki Android WSAN systems	model-based, service choreography, deployment	✓	✗	✗	D-LITe middleware, DSL, BeC^3 tool (consistency check)
[43]	REST \Contiki WSAN systems	model-based, functional design, service choreography, deployment	✗	✓	✗	EMMA middleware, ma- pper (GUI), satisfiability solver
BIP design flow	REST \Contiki WPAN systems	model-based, app design, sys. design, code gener., deployment	✓	✓	✓	DSL, simul., state explor., stat. model checking

concurrent tasks dynamically spawned by the event). Finally, we intend to provide support for additional non-functional requirements, related to energy consumption [35, 53] and security aspects. For the latter, it has been planned to extend the design flow with BIP components that model the security mechanisms of the Contiki OS [54], for being able to identify vulnerabilities and verify the IoT system's protection against node spoofing attacks [55] and denial of service [56] attempts. In another perspective, it is worth to consider using the secureBIP extension [57], as a means for the analysis and synthesis of security configurations in IoT applications that can ensure data and event non-interference [58].

References

[1] Dunkels A, Gronvall B, Voigt T. Contiki-a lightweight and flexible operating system for tiny networked sensors. *29th Annual IEEE International Conference on Local Computer Networks*, IEEE, 2004; 455–462.

[2] Baccelli E, Hahm O, Günes M, Wählisch M, Schmidt TC. RIOT OS: Towards an OS for the Internet of Things. *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2013; 79–80.

[3] Schor L, Sommer P, Wattenhofer R. Towards a zero-configuration wireless sensor network architecture for smart buildings. *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, ACM, 2009; 31–36.

[4] Beal J, Pianini D, Viroli M. Aggregate programming for the internet of things. *Computer* Sept 2015; **48**(9):22–30, doi:10.1109/MC.2015.261.

[5] Hong K, Lillethun D, Ramachandran U, Ottenwälder B, Koldehofe B. Mobile fog: A programming model for large-scale applications on the internet of things. *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC '13, ACM: New York, NY, USA, 2013; 15–20, doi:10.1145/2491266.2491270. URL `http://doi.acm.org/10.1145/2491266.2491270`.

[6] Nastic S, Sehic S, Vögler M, Truong HL, Dustdar S. Patricia – a novel programming model for iot applications on cloud platforms. *Proceedings of the 2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, SOCA '13, IEEE Computer Society: Washington, DC, USA, 2013; 53–60, doi:10.1109/SOCA.2013.48. URL `http://dx.doi.org/10.1109/SOCA.2013.48`.

[7] Sugihara R, Gupta RK. Programming Models for Sensor Networks: A Survey. *ACM Transactions on Sensor Networks (TOSN)* Apr 2008; **4**(2):8:1–8:29.

[8] Castellani A, Bui N, Casari P, Rossi M, Shelby Z, Zorzi M. Architecture and protocols for the Internet of Things: A case study. *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, 2010; 678–683.

[9] Zeng D, Guo S, Cheng Z. The Web of Things: A Survey (Invited Paper). *Journal of Communications* 2011; **6**(6):424–438.

[10] Colitti W, Steenhaut K, Caro ND. Integrating Wireless Sensor Networks with the Web. *Extending the Internet to Low power and Lossy Networks (IP+ SN 2011)*, 2011.

[11] Shelby Z, Hartke K, Bormann C. The Constrained Application Protocol (CoAP). IETF, RFC 7252, 2014.

[12] Cao Q, Abdelzaher T, Stankovic J, Whitehouse K, Luo L. Declarative Tracepoints: A Programmable and Application Independent Debugging System for Wireless Sensor Networks. *Proceedings of the 6th ACM conference on Embedded network sensor systems*, ACM: New York, NY, USA, 2008; 85–98.

[13] Eriksson J, Österlind F, Finne N, Tsiftes N, Dunkels A, Voigt T, Sauter R, Marrón PJ. COOJA/MSPSim: interoperability testing for wireless sensor networks. *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009; 27.

[14] Basu A, Bensalem S, Bozga M, Bourgos P, Maheshwari M, Sifakis J. Component assemblies in the context of manycore. *Formal Methods for Components and Objects*, *Lecture Notes in Computer Science*, vol. 7542, Beckert B, Damiani F, de Boer F, Bonsangue M (eds.). 2013; 314–333.

[15] Sifakis J. Rigorous system design. *Foundations and Trends¬Æ in Electronic Design Automation* 2013; **6**(4):293–362, doi:10.1561/1000000034. URL `http://dx.doi.org/10.1561/1000000034`.

[16] Stachtiari E, Vesyropoulos N, Kourouleas G, Georgiadis CK, Katsaros P. Correct-by-Construction Web Service Architecture. *IEEE 8th International Symposium on Service Oriented System Engineering (SOSE)*, IEEE, 2014.

[17] Nouri A, Bozga M, Molnos A, Legay A, Bensalem S. Astrolabe: A rigorous approach for system-level performance modeling and analysis. *ACM Trans. Embed. Comput. Syst.* Mar 2016; **15**(2):31:1–31:26, doi:10.1145/2885498. URL `http://doi.acm.org/10.1145/2885498`.

[18] Schantz RE, Loyall JP, Rodrigues C, Schmidt DC. Controlling quality-of-service in distributed real-time and embedded systems via adaptive middleware. *Software: Practice and Experience* 2006; **36**(11-12):1189–1208.

[19] Despaux F. Modelling and evaluation of the end to end delay in WSN. Theses, Université de Lorraine Sep 2015. URL `https://hal.inria.fr/tel-01241044`.

[20] Lekidis A, Stachtiari E, Katsaros P, Bozga M, Georgiadis CK. Using BIP to reinforce correctness of resource-constrained IoT applications. *International Symposium on Industrial Embedded Systems (SIES)*, IEEE, 2015; 1–10.

[21] Gubbi J, Buyya R, Marusic S, Palaniswami M. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems* 2013; **29**(7):1645–1660.

[22] Desai A, Gupta V, Jackson E, Qadeer S, Rajamani S, Zufferey D. P: Safe asynchronous event-driven programming. *ACM SIGPLAN Notices*, vol. 48, ACM, 2013; 321–332.

[23] Kovatsch M, Duquennoy S, Dunkels A. A low-power CoAP for Contiki. *MASS'11*, IEEE, 2011; 855–860.

[24] Basu A, Bensalem B, Bozga M, Combaz J, Jaber M, Nguyen TH, Sifakis J. Rigorous component-based system design using the bip framework. *IEEE software* 2011; **28**(3):41–48.

[25] Nouri A, Bensalem S, Bozga M, Delahaye B, Jegourel C, Legay A. Statistical model checking QoS properties of systems with SBIP. *International Journal on Software Tools for Technology Transfer* 2014; **17**(2):171–185.

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

[27] Legay A, Delahaye B, Bensalem S. Statistical model checking: An overview. *Runtime Verification*, Springer, 2010; 122–135.

[28] Hoeffding W. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association* 1963; **58**(301):13–30.

[29] Lekidis A. Design flow for the rigorous development of networked embedded systems. Theses, Universite Grenoble Alpes Dec 2015. URL `https://tel.archives-ouvertes.fr/tel-01261936`.

[30] Abraham D, Alam MS, Duplessis JP, Freeman TW, Hanlon B, Krantz AW, Manchester S, Nick B. XML schema for network device configuration feb " 2" 2010. US Patent 7,657,612.

[31] Halbwachs N, Lagnier F, Raymond P. Synchronous observers and the verification of reactive systems. *Proceedings of the Third International Conference on Methodology and Software Technology: Algebraic Methodology and Software Technology*, AMAST '93, Springer-Verlag: London, UK, UK, 1994; 83–96. URL `http://dl.acm.org/citation.cfm?id=646055.677894`.

[32] Zhou G, He T, Krishnamurthy S, Stankovic JA. Impact of radio irregularity on wireless sensor networks. *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, ACM, 2004; 125–138.

[33] Lekidis A, Bourgos P, Djoko-Djoko S, Bozga M, Bensalem S. Building Distributed Sensor Network Applications using BIP. *Sensors Applications Symposium, 2015. SAS'15*, IEEE, 2015; 1–6.

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

[35] Benini L, Hodgson R, Siegel P. System-level power estimation and optimization. *Proceedings of the 1998 international symposium on Low power electronics and design*, ACM, 1998; 173–178.

[36] Nouri A, Raman B, Bozga M, Legay A, Bensalem S. Faster Statistical Model Checking by Means of Abstraction and Learning. *Runtime Verification*, Springer, 2014; 340–355.

[37] Jan M, Jouvray C, Kordon F, Kung A, Lalande J, Loiret F, Navas J, Pautet L, Pulou J, Radermacher A, *et al.*. Flex-eware: a flexible model driven solution for designing and implementing embedded distributed systems. *Software: Practice and Experience* 2012; **42**(12):1467–1494.

[38] Song Z, Lazarescu MT, Tomasi R, Lavagno L, Spirito MA. High-Level Internet of Things Applications Development Using Wireless Sensor Networks. *Internet of Things*. Springer, 2014; 75–109.

[39] Testa A, Coronato A, Cinque M, Augusto JC. Static verification of wireless sensor networks with formal methods. *Eighth International Conference on Signal Image Technology and Internet Based Systems (SITIS)*, IEEE, 2012; 587–594.

[40] Xu S, Miao W, Kunz T, Wei T, Chen M. Quantitative analysis of variation-aware internet of things designs using statistical model checking. *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*, IEEE, 2016; 274–285.

[41] Di Martino C, Cinque M, Cotroneo D. Automated generation of performance and dependability models for the assessment of wireless sensor networks. *IEEE Transactions on Computers* 2012; **61**(6):870–884.

[42] Cherrier S, Salhi I, Ghamri-Doudane YM, Lohier S, Valembois P. Bec3: Behaviour crowd centric composition for iot applications. *Mob. Netw. Appl.* Feb 2014; **19**(1):18–32.

[43] Duhart C, Sauvage P, Bertelle C. A resource oriented framework for service choreography over wireless sensor and actor networks. *International Journal of Wireless Information Networks* 2016; **23**(3):173–186.

[44] Bertran B, Bruneau J, Cassou D, Loriant N, Balland E, Consel C. Diasuite: A tool suite to develop sense/compute/control applications. *Science of Computer Programming* 2014; **79**:39–51.

[45] Patel P, Cassou D. Enabling high-level application development for the internet of things. *Journal of Systems and Software* 2015; **103**:62–84.

[46] Glombitza N, Pfisterer D, Fischer S. Using state machines for a model driven development of web service-based sensor network applications. *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, ACM, 2010; 2–7.

[47] Taherkordi A, Loiret F, Abdolrazaghi A, Rouvoy R, Le-Trung Q, Eliassen F. Programming sensor networks using remora component model. *International Conference on Distributed Computing in Sensor Systems*, Springer, 2010; 45–62.

[48] Antonopoulos C, Asimogloy K, Chiti S, D'Onofrio L, Gianfranceschi S, He D, Iodice A, Koubias S, Koulamas C, Lavagno L, *et al.*. Integrated toolset for wsn application planning, development, commissioning and maintenance: The wsn-dpcm artemis-ju project. *Sensors* 2016; **16**(6):804.

[49] Shimizu R, Tei K, Fukazawa Y, Honiden S. Model driven development for rapid prototyping and optimization of wireless sensor network applications. *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications*, ACM, 2011; 31–36.

[50] Bliudze S, Mavridou A, Szymanek R, Zolotukhina A. Exogenous coordination of concurrent software components with javabip. *Software: Practice and Experience* 2017; **47**(11):1801–1836.

[51] Poroor J, Jayaraman B. Formal analysis of event-driven cyber physical systems. *Proceedings of the First International Conference on Security of Internet of Things*, SecurIT '12, ACM: New York, NY, USA, 2012; 1–8.

[52] Bouajjani A, Emmi M, Enea C, Ozkan BK, Tasiran S. *Verifying Robustness of Event-Driven Asynchronous Programs Against Concurrency*. Springer Berlin Heidelberg: Berlin, Heidelberg, 2017; 170–200.

[53] Patino MAN. Energy efficiency in data collection wireless sensor networks. PhD Thesis, Purdue University 2016.

[54] Halcu I, Stamatescu G, Sgârciu V. Enabling security on 6lowpan/ipv6 wireless sensor networks. *7th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, IEEE, 2015; SSS–29.

[55] Basagiannis S, Katsaros P, Pombortsis A. An intruder model with message inspection for model checking security protocols. *Computers & Security* 2010; **29**(1):16 – 34.

[56] Deshpande T, Katsaros P, Basagiannis S, Smolka SA. Formal analysis of the dns bandwidth amplification attack and its countermeasures using probabilistic model checking. *2011 IEEE 13th International Symposium on High-Assurance Systems Engineering*, 2011; 360–367.

[57] Said NB, Abdellatif T, Bensalem S, Bozga M. A model-based approach to secure multiparty distributed systems. *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, 2016; 893–908.

[58] Fernandes E, Paupore J, Rahmati A, Simionato D, Conti M, Prakash A. Flowfence: Practical data protection for emerging iot application frameworks. *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association: Austin, TX, 2016; 531–548.

Appendices

A *Language for Contiki REST application design definition (DSL)*

This Appendix refers to the DSL syntax and BIP templates, for client actions, which are not mentioned in Section 3.1. Moreover, a server process in DSL and its Contiki template are presented.

```
<while boolExp="bool-expr">
  action+
</while>

<if boolExp="bool-expr">
  action+
  <elseif boolExp="bool-expr">*
    action+
  </elseif>
  <else>?
    action+
  </else>
</if>

<wait>
  <onEv evType="event-type"?
      cond=""? >*
    action+
  </onEv>
</wait>
```

Listing 5: DSL syntax for structured actions

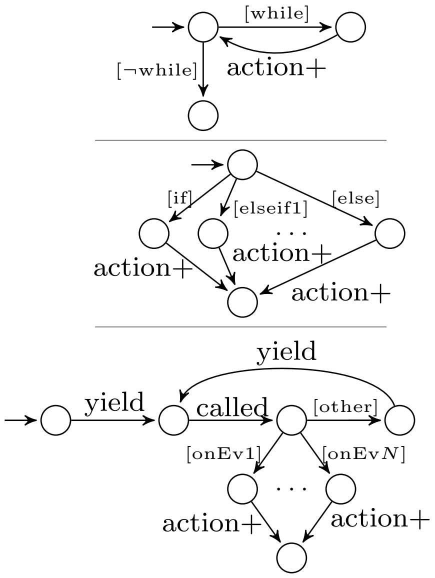


Figure 16: BIP templates for actions in Listing 5

The actions can encode control flow structures with nested actions, such loops, conditionals or event handlers. We call these actions *structured* and their syntax is defined in Listing 5. Moreover, Figure 16 shows the corresponding BIP template for each of them.

```
<timeout timer="QName"
  command="set|rset|rstrt|stop"
  (var="QName"|val="unsign-int")? />

<postEv mode="syn|asyn"
  evType="event-type"
  process="QName"?
  var="QName"? />

<sndReq server="QName"
  resource="QName"
  params="QName-list"?
  method="put|get|post|del"
  (contentType="txt|json|xml"
    var="QName")? />

<getResp/>

<exit />

<code> <!--C/C++ code--> </code>
```

Listing 6: DSL syntax for basic actions

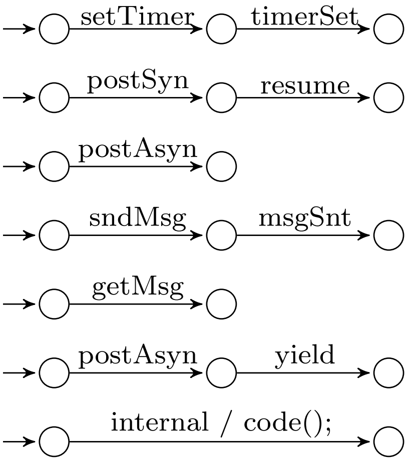


Figure 17: BIP templates for actions in Listing 6

A set of *basic* actions including timeout, event dispatching, message communication, block of code etc. have been also encoded, whose syntax is presented in Listing 6. The BIP behaviour that is instantiated for these actions is shown in Figure 17, where each action is modelled by the activation of one or two successive ports. For a <timeout>, the process first sets a timer (*timerSet* port) and waits until the timer is set by Contiki (*timerSet* port). The <postEv> action corresponds either to a synchronous or an asynchronous posting. After synchronous posting (*postSyn* port), the process is blocked until be allowed to resume (*resumr* port) by the Contiki. This is in contrast with the asynchronous posting, which does not block the process. With the <sndReq> action, the process creates a message to be sent by Contiki (*sndMsg* port) and it is notified when the message is sent (*msgSnt* port). The <getResp/> action (*getMsg* port) is used when receiving a response message, whereas with the <yield/> action (*yield* port) the process yields. With an <exit/> action, the process posts asynchronously an EXIT event for itself and yields. The process is called, when the EXIT event is scheduled to occur and the exit handler is then triggered. Except from the aforementioned actions, the DSL provides also the <code> element that allows for custom actions which are specified using C/C++ code. In BIP, this element is represented by an internal action (no port is activated) that calls the function with the specified code.

A server process description (Listing 7) includes a set of (periodic or aperiodic) resource handlers, with handlers for their supporting COAP methods. The "autoStart" value[8] defines whether the process will be initiated by default, or upon an event by another process. Listing 8 shows the Contiki syntax of a server process. At the bottom of the template (lines 13-19), there is the definition of a process, which lives only to start a REST engine and activate the resources. The REST engine (whose code is included in the server's source file) is a predefined Contiki process that implements a REST server, i.e. it invokes the server's resource handlers either periodically, or upon an incoming request. The communicated messages are passed between the REST engine and the handlers with the *req* and *resp* variables. The template includes two resources, one aperiodic (line 2) and one periodic (line 6). Resource handlers are implemented as a `[resource_id]_handler` function.

```
<server id="..."
 autoStart="[true|false]"? >+
  <resource_handler
     resource_id="..." >+
   <method id="[get|post|put|del]{1,4}"/>+
   <code> <!--C/C++ code--> </code>
   <periodic period="int" >
     <!--C/C++ code-->
   </periodic>?
  </resource_handler>
</server>
```

Listing 7: DSL syntax for a REST server

```
 1 RESOURCE(res1,/*methods*/,...);
 2 void res1_handler(REQUEST* req,
 3                                 RESPONSE* resp)
 4 { .... }
 5
 6 PERIODIC_RESOURCE(res2,/*period*/,...);
 7 ... /* res2 handler */
 8 void res2_periodic_handler(REQUEST* req,
 9                                 RESPONSE* resp)
10 { .... }
11 /* other resources and handlers */
12
13 PROCESS(server, ...);
14 AUTOSTART_PROCESSES(&server);
15 PROCESS_THREAD(server, ... ){
16   PROCESS_BEGIN();
17   .... /* start rest engine process */
18   .... /* activate resources */
19   PROCESS_END();}
```

Listing 8: Contiki code template for a REST server

B *BIP interactions of the RestModule model with the OS model*

In <AppModel>, each <RESTModule> interacts with the <OS> component as it is shown in Figure 18. For simplicity, only the OS interactions with one process are depicted. Every process is modelled as an atomic component with application-specific behaviour.

A process is called (*called* port), when the OS dispatches an event to it. After the event is handled, the process's execution yields (*yield* port). Posted synchronous or asynchronous events to other processes are passed through the *postSyn* and *postAsyn* ports. For a synchronous event, the process resumes execution (*resume* port) upon the end of event handling. Each process may request polling for itself or other processes and can set deadlines using timers (*setTimer* port) or send a message (*sndMsg* port). The <ConKernel> acknowledges the completion of setting a timer or transmitting a message (*timerSet*, *msgSnt* ports). When the process execution is finished, the *end* port is enabled. The model details for the invocations of REST resource handlers are discussed in [20]. In current model, the supported resource types are periodic, event and actuator.

C *Network stack model parameters*

The model parameters in Table 6 can be adjusted through the network configuration XML specification (input in step 2). Some parameters concern with the exponential backoff mechanism of the IEEE 802.15.4 standard or the timeout for a packet receipt. Moreover, there are parameters like

[8] The "autoStart" attribute is used for both servers and clients, although it was not shown in Section 3.1

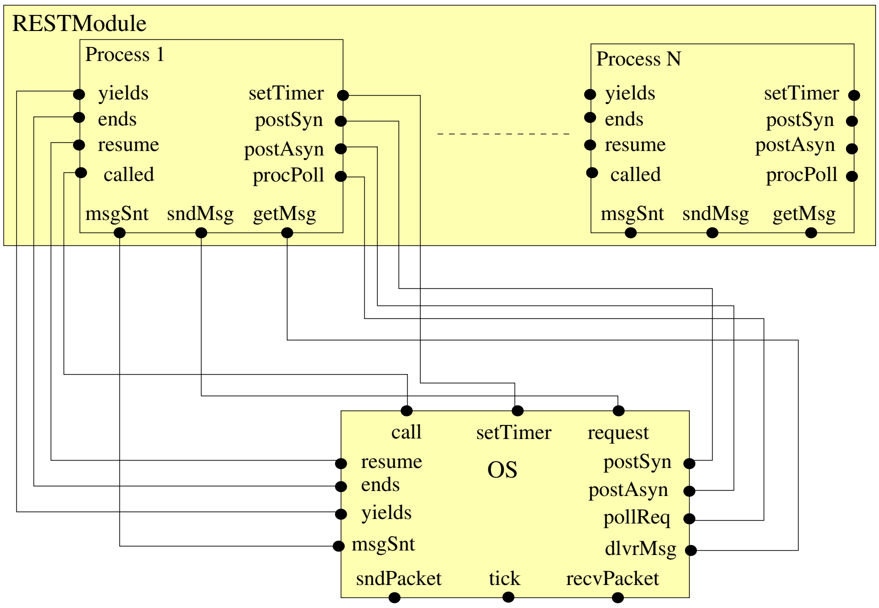


Figure 18: The RestModule for a Client and its interactions with the OS

macMinBE, macMaxBE, macMaxCSMABackoffs and *macMaxFrameRetries* that affect the network throughput and the number of channel collisions. Parameter values depend on the transmission time of one symbol (4 bits), denoted by *symbolPeriod*. This time is computed from equation (1).

Model parameter	**Value**
aUnitBackoffPeriod	$20 * symbolPeriod$
CCA duration [1]	$8 * symbolPeriod$
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 [2]	$12 * symbolPeriod$
LIFS [2]	$40 * symbolPeriod$

Table 6: Parameters of the modelled network stack

[2] Short Interframe Space (SIFS) is the period required for allowing the MAC layer time to process the data received in the physical layer for short data frames and Long Interframe Space (LIFS) is the respective period for long data frames

[1] Clear Channel Assessment (CCA) is the time needed to access the communication channel