Compiling Business Process Models for Sensor Networks

Alexandru Caracaş[‡], Alexander Bernauer[*]
[‡]IBM Research - Zürich [*]ETH Zürich, Institute for Pervasive Computing

***Abstract*—Wireless sensor networks are increasingly being used to improve business processes. The behavior of such a process is usually captured in models while its implementation is typically created manually. Besides being expensive, this approach entails deviations between the model and the implementation of a business process, thus rendering it less effective. We aim at closing this gap by automatically generating applications from the model of the process instead. In our approach, software developers provide building blocks that are used by business analysts to model business processes and hereby effectively control their execution on an abstract level. Furthermore, the model editor integrates a compiler and a simulator so that the business analysts can test and debug the modeled processes on the same level of abstraction. Our evaluation results show that the generated code can be executed on resource-constrained sensor nodes consuming only 1% more energy than the hand-written equivalents. However, the benefits of our approach come at the price of 10% more RAM and 44% more flash space consumption on average.**

I. Introduction

Business processes drive the manufacturing of products and the operation of services in today's economy. They describe the interactions between humans, IT-systems, and the environment that are necessary to reach the business goals. With increasingly evolving sensor, wireless and distributed systems technologies, more and more business processes use *wireless sensor networks (WSN)* to gather information about or interact with the entities they control. Examples include safety processes for storing hazardous materials [1], securing shipping and handling of containers to comply with government regulations [2], and automating hospital processes [3].

Business processes are designed and modeled by business analysts who are domain experts in their field but do not necessarily have a background in IT. Thus, the actual implementation of a business process is typically performed by software developers. This prevalent approach poses two major problems. Firstly, the implementation usually deviates from the intentions and expectations of the business analysts because cross-domain communication is prone to misunderstandings. This deviation typically renders the respective business process less effective the larger it becomes. In order to avoid this, technology implementations should follow the business process and not vice versa [4]. Secondly, changing the model of a business process necessitates the adaption or even recreation of parts of the implementation. As this is expensive and time-consuming [5], business processes tend to adapt slowly to changes of requirements or business facts.

In this paper, we propose to use the standardized *Business Process Model and Notation (BPMN)* [6] as the interface language between business analysts and software developers and to apply code generation to turn BPMN models into executable code. BPMN models are refined by additional models while commonly programmed components form the base case of this recursion. These components and the code which is generated from the models constitute the final application. Software developers can decide which levels of abstraction should be modeled and which should be programmed. At the topmost levels, business analysts, which represent our target user group, model the business process in the familiar BPMN while software developers relieve them from mastering distributed computing problems by implementing all models and components below.

In our approach, instead of being just a documentation blueprint, a BPMN model actually defines the execution of the respective business process and can be simulated and debugged directly in the model editor. Additionally, BPMN supports the integration of multiple involved distributed systems into a single top-level model, thus covering the demands of business processes which use WSN. To enable better integration into the environment, reduce power consumption, and decrease production and deployments costs [7], we focus on the lower-end spectrum of embedded devices such as 8-bit MCUs with 128 kB of non-volatile memory, and 8 kB of volatile memory. To this end, our compiler generates efficient code which runs on such resource-constrained sensor nodes.

The quantification and evaluation of modeling and programming abstractions is notoriously difficult, but we make a series of arguments to support our approach. Firstly, BPMN is an established standard and business analysts already use it to model business processes. Hence, it is an adequate tool for such tasks and the addressed user group is capable of using it. Next, we show that attaching execution semantics to BPMN models is possible. We do this both in theory and in practice. Firstly, by discussing how the code generation works and secondly, by presenting results of a series of case studies for which we used a proof of concept implementation of the

[*]This author has been partially supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322, and by CONET, the Cooperating Objects Network of Excellence, funded by the European Commission under FP7 with contract number FP7-2007-2-224053. [‡]This author would like to thank Thorsten Kramp and the Mote Runner team for their support and guidance.

required tools. Additionally, we demonstrate that the generated code is efficient enough to be executed on resource-constrained sensor nodes. Last, we argue that simulating and debugging the models of business processes is an intuitive task for business analysts because this functionality is integrated into the model editor. In summary, we enable business analysts to control, simulate, and debug the execution of their business processes on an abstract level.

This paper is organized as follows: Section II presents related work and argues how our approach advances the state of the art. Next, Section III gives a brief introduction into the BPMN notation and semantics using an example business process. We then outline how BPMN models can capture WSN requirements in Section IV, and show how WSN applications can be generated from BPMN models in Section V. Our modeling approach requires an integrated tool-set which we describe in Section VI while in Section VII we stress the debugging aspect. Subsequently, we evaluate our proof of concept implementation of the tools in Section VIII using a representative set of WSN applications before concluding in Section IX.

II. Related Work

Probably the best-known modeling language is the Universal Modeling Language (UML) [8] which is a generic set of standard graphical notations that capture different system aspects. Rhapsody [9] is an industry solution for UML-driven development and automatic code generation for embedded systems. Recent work in this area even addresses WSN [10] using Sun SPOTs [11] with a 32-bit ARM core. In comparison, our solution supports sensor nodes with significantly less memory and computation resources. Additionally, for the description of a single business process, UML requires multiple models which cover different aspects and different involved entities of the business process. This reflects the fact that UML is a rather technical notation which does not fit the needs of business analysts.

Other proposed solutions for graphical programming and model-based development of sensor network applications [12], [13] are based on the LabVIEW[14] and Simulink[15] proprietary standards. Additionally, Viptos [16], Flow [17], and Srijan [18] are modeling environments for wireless sensor networks with custom graphical notations based on data flows. In either case, the intended audience again are scientists, system engineers, and software developers rather than business analysts.

To address the needs of the business domain, the Business Process Model and Notation (BPMN) [6] became an Object Management Group (OMG) standard for business processes. Today, this language is widely used by business analysts, consultants, and executives as a lingua franca. BPMN is additionally suitable for transformation to execution languages such as the Business Process Execution Language (BPEL) [19] where a BPEL engine executes models while communicating with the environment via Web Services. Existing approaches which integrate sensor networks into business processes focus on this approach [20], [21], [22]. For the resource-constrained sensor nodes which we want to support, a model interpreter like a BPEL engine is not a viable option. Instead, we use a compiler to generate efficient code which executes directly on the targeted sensor node platform. Furthermore, with our approach, different communication channels can use different protocols such as IEEE 802.15.4 or TCP/IP depending on the respective requirements.

In summary, none of the existing solutions bridges the gap between the model description provided by business analysts and the corresponding implementation of a business processes for the resource-constrained sensor networks we are targeting. On the one hand, although some of the modeling solutions may accommodate the resource requirements, they expect the business audience to learn and understand a technical language and the associated tools. On the other hand, solutions targeted particularly at the business domain require too many device resources. We combine the benefits of existing work and use an open industry standard for modeling business processes. Our approach provides a unified view over all involved systems, supports simulation and debugging of business processes directly in the model view, and generates efficient code.

III. Business Process Example and Background

This section introduces the reader to a subset of BPMN terms and concepts while we refer to the standard [6] for an exhaustive description. Generally, a BPMN model describes a process with the interactions and behaviors of involved entities which cooperate in order to fulfill required business goals. In BPMN terminology, a *pool* represents such an involved entity and acts as the topmost container for its model. Its graphical representation is a rectangle which can be collapsed to visually compress a model.

Figure 1 shows an example BPMN process for parcel delivery with three pools: `Headquarters`, `Truck`, and `Parcel`. Only the `Headquarters` pool is expanded and shows the details about enclosed tasks, control flows, message flows, and data flows. This is an example of a classical BPMN model as there is no code generation attached but the tasks are performed *manually* instead.

The actual work performed by a pool is divided into *task*s which are shown using boxes with round corners. A task may be recursively refined to contain other models in which case a plus-sign annotation is placed in the lower part of the box. Otherwise, the task is a *basic task* which is implemented directly in the language of the target platform.

Changes in the control flow are described using a *gateway* rhombus with an enclosed symbol which defines the semantics. A *parallel gateway* has a plus sign and forks the control flow if there are multiple outgoing sequences and joins the control flow if there are multiple incoming sequences. An *if gateway* has an X-symbol and represents a conditional decision point where the process follows the respective sequence flow that matches the condition.

During execution, a process may wait for a certain *event* to occur (*catch*) or it may trigger (*throw*) events to notify

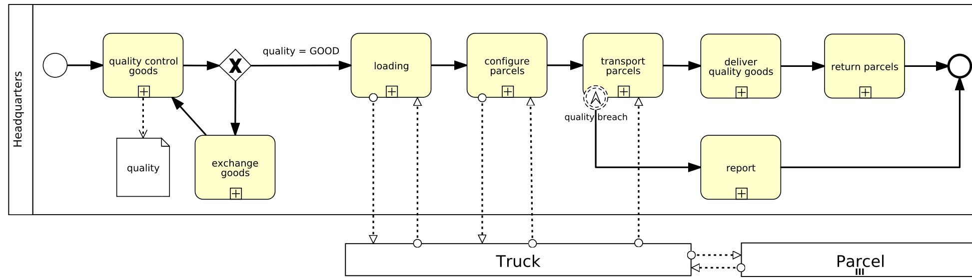

Figure 1. Classical business process which delivers parcels while monitoring that the environmental temperature stays within allowed thresholds to ensure the quality of the delivered goods. The process involves the headquarters, a truck, and parcels. The truck ships parcels from the headquarters to a destination and each parcel contains goods and a WSN node which monitors the temperature. In the truck there is a base node equipped with mobile Internet connectivity which communicates with both the parcels and the headquarters. This business process is intended to be part of the transport company's overall business process and is executed for each set of goods which has to be delivered to a specific destination by a single truck.

different tasks or other pools. Examples of possible events include starting and ending a task, receiving or sending a message, and signaling exceptional conditions.

The BPMN representation for events is a circle. To visually distinguish between the different event types and their properties, BPMN uses icons and different styles for the circle line. Catch events have a hollow icon, while throw events have a filled icon. Furthermore, start events have a thin border and end events have a thick border. A catch event that does not terminate the respective process has a dashed line, e.g. `quality breach` in Figure 1. In contrast, other catch events such as `damaged` in Figure 2 do terminate the respective process and are depicted with continuous lines.

The data objects required by a process to fulfill its work can be explicitly represented in BPMN using *data items* (papers folded on the upper-right corner) and *data stores* (database cylinders). Both are connected to readers and writer by means of *association flow* arrows which are depicted as a dotted arrow. The difference between data items and data stores is that the former are volatile while the latter are persistent.

Communication between pools is explicitly modeled using *communication flows* which are represented as dotted lines with a hollow arrow head on the receiver side and an empty circle at the sender side. An envelope on the communication flow arrow marks the actual definition of the communication and specifies its properties.

Figure 2 shows the model of the `Parcel` pool. This process is automatically executed on the WSN nodes and demonstrates one of the major contributions of our work: The business analysts define reactions to business events on an abstract level and hereby control the actual execution of the business process because their model will be used to generate the corresponding application that is executed on the sensor node.

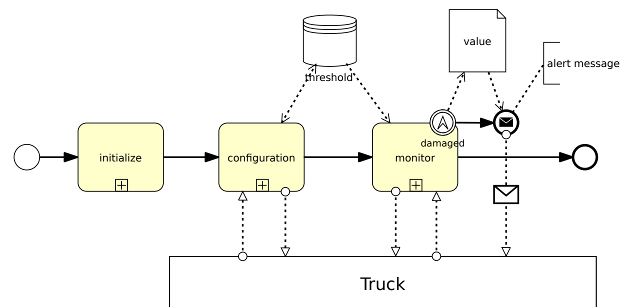

Figure 2. Model of the `Parcel` which is created by business analysts. After initialization and configuration, the process starts the `monitor` sub-task. In case of a `damaged` event, a message is sent to the truck. Otherwise, the process simply ends as soon as `monitor` completes. Code generated from this model is executed on a WSN node.

IV. BPMN Models for WSN Applications

Models for WSN applications should support the distributed, heterogeneous and reactive characteristics of this domain. For energy efficiency considerations, models should also allow control over the protocols used for communication. Additionally, since a WSN is usually connected with other systems, the modeling language should support such integration.

The syntax and semantics of BPMN need neither be extended nor restricted to fulfill these requirements. The reactive nature of WSN applications is captured by BPMN events and the processing of events by BPMN sequence flows. Furthermore, communication only takes place via messages between pools and pools can have multiple instances. Thus, different entities of a distributed system such as various kinds of WSN nodes can be modelled with different pools.

The visual integration of the different entities is provided by the fact that pools can be collapsed and integrated into a system-wide model which shows only the general communication flow between the pools. Moreover, a definition of a communication flow has an envelope component attached to it and the properties of this component defines the details of the employed communication protocol. Thus different transmission modes such as broadcast and unicast, and different communication protocols such as IEEE 802.15.4 and TCP/IP can be chosen for each communication flow separately.

By this means, manually performed and automatically executed business processes can also be integrated into a system-wide model. To this end, base stations could send emails to

an operator who sends back commands via HTTP requests, for instance. The communication between the operator and the WSN is simply represented with properly annotated communication flows in a BPMN model and the code generation takes care that the communication routines which are provided by the software developers are invoked.

In BPMN, each pool contains a task which uses sub-tasks. Such a sub-task can be refined by an additional model or can be a *basic task* which maps to an API function of the *target platform*. In the end, every sequence flow results in a sequence of basic tasks being called. The set of all basic tasks is part of the *platform API* which the software developers implement once and reuse for all business processes which involve the given target platform.

While graphical modeling provides a better overview over a process and is more intuitive than programming, it is not a suitable tool for all jobs. In particular, software developers generally prefer writing code when dealing with low-level implementations. Due to the recursive nature of BPMN models, our approach allows to choose the border between modeling and programming at arbitrary levels of abstraction. Hence, the *building blocks* for the models which are created by the business analysts can contain both basic tasks and tasks which are modeled. As a rule of thumb, general sequence flows of processes should be modeled while algorithms and device drivers should be programmed.

V. Code Generation

The core of our approach is a compiler which translates models into executables. In this section, we describe the code generation part of this translation which takes a sound BPMN model as input and yields an *intermediate representation* which is further processed by later steps. A sound model does not have a "lack of synchronization" and is "deadlock-free" as defined in [23]. This basically means that it is not possible that two sequence flows execute the same part of the model at the same time, and that there is always at least one sequence flow executing.

The translation of association flows is straightforward. Data items become variables and access to data stores is mapped to calls of persistence functions provided by the platform API. This translation is correct as long as no task is ever invoked more than once at the same time. We call this the *static* mode because it provides a one-to-one mapping between tasks and its enclosed data objects.

In contrast, in the *dynamic* mode there are *parallel tasks* which are invoked more than once at the same time. Then each invocation of such a task needs its own set of data objects. Thus, in the generated code, the corresponding variables are dynamically instantiated with each invocation and access to them is performed indirectly.

It is undecidable in general if a task is parallel or not. However, it is possible to identify most non-parallel tasks by a simple sequence flow analysis. For the undecidable cases the dynamic mode is always safe but it might be needlessly inefficient. In these cases the user might be able to assist by annotating tasks as non-parallel which is a property that can be checked at runtime by the generated code.

In contrast to the associations flow, the translation of the sequence flow is more complex. To support standard sensor nodes we have to address the fact that major WSN platforms are event-driven due to the efficiency benefits of this paradigm [24], [25], [26]. Thus, the compiler has to translate the sequential and parallel computation semantics of a BPMN model into equivalent event-based code.

Event-based platforms typically use a single *dispatcher* which constantly takes the oldest *event* out of an event queue and calls the corresponding *event handler*. Event handlers may not block when waiting for something. Instead they only trigger a corresponding *split phase operation* and delegate the remaining processing to a *callback* which is a second handler that is registered at the runtime environment.

By following this pattern the compiler maps each sequence flow to a chain of event handlers which are causally related to each other by the respective events. As long as a split-phase operation has not yet completed, the dispatcher calls handlers which correspond to other sequence flows, thus processing multiple sequence flows virtually in parallel.

We call a basic task with a split-phase interface a *critical task*. Additionally, all tasks which enclose critical tasks are also called critical. Outgoing sequence flows of critical tasks, catch events, start events, receive events, and outgoing sequence flows of parallel gateways represent all possible start or resume points of sequence flows at runtime. Thus, for each instance of these entities a function is generated which executes the corresponding sequence flow.

If a sequence flow reaches a task, the generated code simply calls the function corresponding to the start event of the invoked task. Furthermore, for every possible continuation - i.e. catch events at the border of the task or outgoing sequence flows in the case of a critical task - the pointers of the corresponding functions are saved in extra *continuation variables*. These variables are managed the same way as data item variables are.

If a sequence flow reaches a throw event, the generated code simply consults the corresponding continuation variable and calls the continuation. If a sequence flow reaches an end event, the generated code simply returns from the function in the case of a non-critical task. Otherwise it consults the continuation variable for the outgoing sequence flow and calls the continuation. Basic critical tasks are manually implemented along these lines, so that the computation can continue after completion of a split-phase operation.

Terminate events require more effort as the BPMN semantics require that all active sequence flows of the enclosing task are canceled. As sequence flows always continue execution until they reach a critical basic task, canceling them simply requires to suppress the invocation of the callbacks. Thus, for every critical task a *terminate function* is generated which recursively invokes the terminate functions of the enclosed subtasks. In the case of critical basic tasks, the terminate function is manually implemented and suppresses the invocation of the

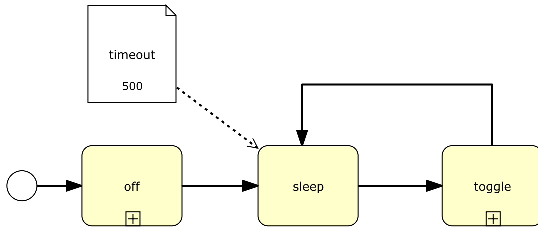

Figure 3. Model for a *Blink* application which toggles an LED every 500 ms.

callback in a platform-specific way.

The transformation of if gateways simply generates a corresponding if structure. Concerning parallel gateways, the translation exploits the fact that in a sound model there is always a one-to-one mapping between fork and join gateways. Note that for this to be true one has to consider an implicit join gateway in front of all end events. Furthermore, every throw event is implicitly followed by an end event. These model transformations are performed automatically and do not alter the model semantics.

Each pair of join and fork gateways share a common *counter variable* for the number of active sequence flows between the fork and the join. So, the translation of the fork gateway is code which first increases the counter by the number of outgoing sequence flows and then calls the respective functions of the outgoing flows in turn. Accordingly, a join gateway results in code which decrements the counter by one. If and only if the counter equals zero the function that covers the single outgoing sequence flow is called.

Listing 1. Sample generated code in static mode from the *Blink* model.

```
1   void (*continue_sleep)();
    int blink_timeout = 500;
    int toggle_status;
4
    void start_blink() {
        start_off();
7       continue_sleep = &blink_after_sleep;
        sleep_timeout = blink_timeout;
        start_sleep();
10  }

    void blink_after_sleep() {
13      start_toggle();
        continue_sleep = &blink_after_sleep;
        sleep_timeout = blink_timeout;
16      start_sleep();
    }

19  void start_off() {
        led_setStatus(0);
    }
22
    void start_toggle() {
        toggle_status = led_getStatus();
25      toggle_status = ! toggle_status;
        led_setStatus(toggle_status);
    }
```

Listing 1 shows the transformation of the model in Figure 3 in static mode. The tasks `off` and `toggle` are non-critical sub-tasks because they only invoke the non-critical basic tasks `led_setStatus` and `led_getStatus` (line 20, 24 and 26; not shown in the model). In contrast, `sleep` is a critical basic task. Thus the continuation `blink_after_sleep` is stored in the continuation variable `continue_sleep` before the task is invoked (line 7 and 14). Furthermore, the `timeout` data item is associated with the `sleep` task, so that its value is passed to the sub-task (line 8 and 15).

VI. Tools

The realization of our approach requires a set of integrated tools: an editor, a compiler and a simulator.

The compiler transforms a model into an executable in a three-phase translation. In the first phase, the model is checked for consistency and soundness. Next, the compiler performs a pattern-based platform neutral translation as described in Section V. The third phase then is platform specific and translates the intermediate representation from phase two into suitable code for the target platform. For each entity, the compiler generates one binary using a back-end for the respective target platform. These binaries can be installed on real hardware or executed in the simulator.

There are several constraints on the model which should be checked by the compiler in phase one. For example, unconnected flows, unused data or undefined tasks are not allowed, and all thrown events must also be caught. Furthermore, a type checker should ensure that data which is associated with a basic task has a type which is suitable to what the task expects. Finally, software developers should be able to implement further checks in order to ensure the correct usage of the building blocks they provide. For example, certain basic tasks might only be allowed to be called once or it might be compulsive that other basic tasks are invoked in a certain order.

As mentioned previously, phase three is platform specific. As the concepts of the intermediate representation are very basic, we expect that it is possible to implement them on different target platforms. The standard compiler tool chain of this platform finally processes the output and links it against the implementation of the platform API and other required platform libraries in order to produce the final binaries.

Compilation and simulation are triggered by the business analysts who use the editor as their primary tool. It allows them to create, browse, and manipulate business process models. Furthermore, it offers means to manage, create, and import model libraries. What distinguishes this editor from all other BPMN editors, though, is the seamless integration of the compiler and the simulator. The editor controls these tools by sending commands and receiving resulting information which are visualized in the model. For example, in case of a constraint violation the compiler propagates the information about the constraint and the violating BPMN component back to the editor. And in case of compilation or runtime errors, the compiler or the simulator can infer the causing BPMN component (c.f. Section VII) so that the editor can then highlight this component in the model.

In the overall development cycle, there are different kinds of possible failures. Constraint violations indicate errors in the model which the business analysts have to correct. In contrast, compilation and runtime errors are the responsibility of the

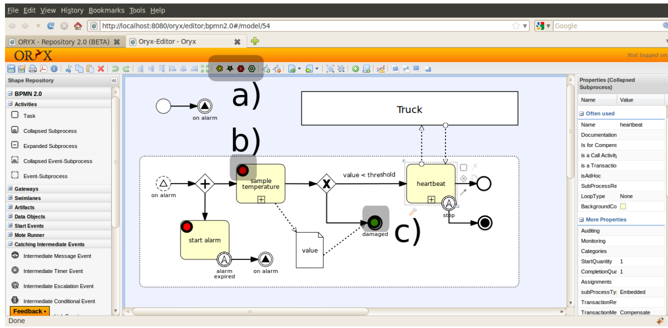

Figure 4. Web-based model editor for BPMN modeling, simulation and debugging of wireless sensor networks showing: a) buttons for compile, run, toggle breakpoints, and stepping, b) active break-point, c) break-point hit.

software developers. Either by changing the implementation of the platform API or by adding additional constraints, they have to ensure that a model which passes all constraint tests does not cause such failures. This is important because business analysts typically cannot and should not handle compiler error messages and stack traces. Last, runtime exceptions can occur as part of a valid process and are represented as events in BPMN models where they can be caught and handled. In general, we recommend to avoid the usage of exceptional events in higher level models because they introduce a non-sequential control flow which is sometimes difficult to follow for people with little background in IT.

Section VIII presents a proof of concept implementation of the above described tools. Before that, the following section elaborates on the debugging support, which is one of the major contributions of our work.

VII. Debugging

A major benefit of a modeling or programming abstraction is the concealment of lower-level details. Without being forced to understand and master all those details, a user of the abstraction can create and maintain applications faster and more intuitively. This benefit is undermined if the application contains an error and there is no debugging support on the application's level of abstraction. In such a case, the user has to infer the error in the application code from the details of the generated code. Many WSN programming abstractions suffer from the lack of debugging support [27], which we believe hinders their adoption in practice.

Consequently, we want to support debugging on the abstraction level of the BPMN models. To this end the compiler generates a *symbol table* during phase two. This symbol table maps all components of the input model to locations in the generated code. This information can be used by the tools to retrace the translation between the model and the executable in both directions.

Concerning the tools, debugging support means that everything should be integrated into the editor which is the primary tool for the business analysts. They should be able to set breakpoints at arbitrary points of sequence flows either for a single instance or for all instances of an entity. If such a breakpoint is hit during the runtime of the binaries, for each instance the current progress of all sequence flows and the current value of all data objects should be visualized in the editor. Furthermore, it should be possible to alter the value of data objects and to inject events so that different sequence flow paths can be tested.

A simulator which supports these features must be able to simultaneously execute all instances of all entities. If one instance hits a breakpoint, the execution of all instances has to be stopped so that the editor can query the progress of sequence flows and the value of data objects. To this end, the symbol table helps to map between graphical components and points in the generated executables.

Simulating and testing business processes under various possible conditions gives confidence to the business analysts that the created business process will work as intended once it is deployed. For that, it is crucial that the simulation environment matches the real environment of the business process as close as possible. Furthermore, the models and the basic tasks should not contain non-deterministic behavior because this cannot be reliably tested with the described simulation setup. We suggest that the software developers ensure this by creating models and basic tasks accordingly and by adding constraints to the compiler.

VIII. Evaluation

To evaluate our model-driven methodology, we implemented the required tools as described in Section VI. Our proof of concept implementation supports all described requirements except for changing data object values in a stopped simulation. Furthermore, injecting external events is not integrated into the model editor but has to be done by means of command line tools.

Figure 4 shows our model editor which is a web-based application based on Oryx [28], a BPMN 2.0 editor distributed under the MIT license. We extended the functionality of the Oryx editor to include plugins that integrate the compiler and the simulator into the editor. The communication interface between the web browser, our compiler toolchain, and the sensor network simulation framework uses a RESTful API with data encapsulated in JSON messages.

We use the Mote Runner [26] platform as WSN development environment. The platform hides the intricate details of the underlying hardware by using an efficient virtual machine and micro-manages power on behalf of applications. Applications are programmed using high-level languages which are further transformed to optimized byte-codes. These features relieve our compiler from hardware-specific details so that it can generate portable high-level code. Mote Runner additionally provides a simulation environment with a source-level debugger interface which our integrated editor exploits to simulate and debug the same application byte-codes that are executed on real sensor nodes.

The compiler plugin calls the compiler tool which generates C# source code for the Mote Runner platform. The simulator plugin creates the corresponding sensor network configuration

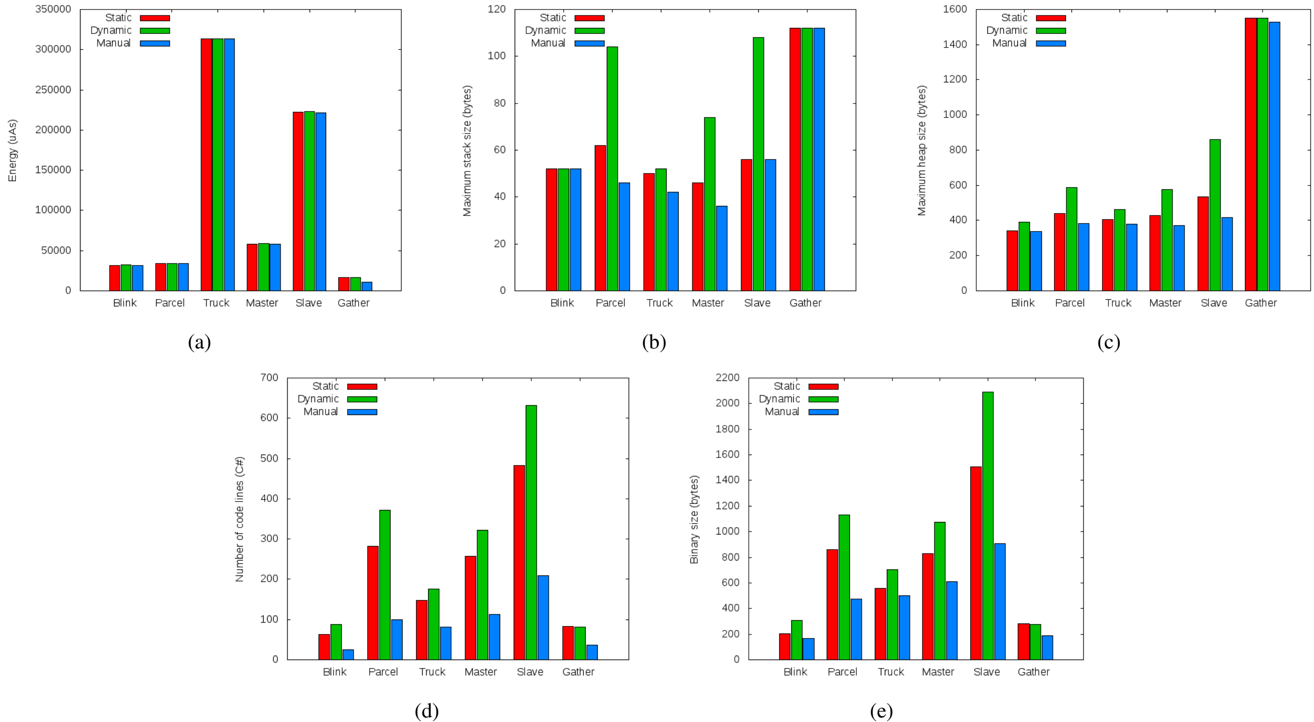

Figure 5. Comparison of generated code (static and dynamic) against the manual equivalent in terms of: energy consumption (a), RAM usage for heap (c) and stack (b), lines of code (d), and flash usage (e) of the binary application. The size of the dynamic BPMN run-time library is only 724 bytes and is included in the measurements for the dynamic case.

for the simulator according to the pools and their respective cardinality. With the aid of the symbol table from the compiler, errors and information from all tools are processed and reported back to the modeler in the browser with warning or error indicators overlaid on the graphical symbols.

To show the generality and expressiveness of the approach, we used our editor to model several applications corresponding to several WSN archetypes [29]. Firstly, we refined the parcel monitoring scenario described in Section III using a single-hop star topology with a *Truck* and several *Parcels*. Secondly, we modeled a multi-hop tree-based TDMA protocol where the *Master* root sends beacons which are re-broadcasted in assigned slots by *Slave* nodes. This shows that we can even express low-level communication and synchronization details. In addition, we modeled *Gather* which is a high-level data collection application using a decentralized multi-hop library for the communication. Last, we modeled the base case *Blink* which toggles an LED every $500\,ms$ to clearly expose the systematic overhead of our approach.

To quantify the efficiency of our approach we measure the usage of resources of the generated code against the hand-written equivalents. To this end, we compiled each application once in static and once in dynamic mode with all tasks treated as parallel tasks (c.f. Section V). This is only possible because in none of our examples we encountered the need of this feature. Nevertheless, we think it is interesting to know the costs of supporting all BPMN features.

The total amount of consumed energy is an important efficiency measurement because the life time of battery-powered sensor nodes depends directly on it. The maximum stack size and the maximum heap size show how efficiently RAM resources are used while the size of the binary does the same for the flash resources. Finally, the code size is a common indicator for code complexity. All measurements were obtained using the simulation environment running for 30 seconds which is enough time to measure and explore both the normal sequence flows as well as the exceptional code paths.

Once a process behaved in the simulation as expected, we used Iris [30] sensor nodes which have an 8-bit micro-controller, 128 kB of flash and 8 kB of RAM for a test deployment. We did not measure the resource consumption on the real hardware because the Mote Runner simulator provides accurate measurements by design. Instead, we thereby only verified that the generated code is efficient enough to be executed on such a resource-constrained device.

Figure 5 shows the results of the evaluation. First, we see that the energy consumption overhead for each compilation mode is only 1%. This is because WSN applications are reactive by nature, where sleeping periods dominate computations.

Second, the overhead for the RAM usage averages at 10% in the static case and 50% in the dynamic mode. In contrast, the size of the binary and the code show that the systematic transformation of the model results in twice more code than a software developer needs for the same task. On average, the overhead for the flash consumption is 44% for the static mode

and 3 times more space for the dynamic mode.

In practice, the energy consumption is the most important efficiency characteristic of an application as long as the employed mote provides enough resources to host it. As we could show that the generated code fits on a typical mote and the additional energy consumption is very low, we argue that the benefits of the code generation outweigh the introduced overhead. This is especially true for the static mode for which the overhead is moderate given that flash space is usually not the major limiting resource on motes.

Concerning the dynamic mode, it is currently not clear if parallel tasks are used rather often or rather seldom when modelling WSN applications. All we can say is that the total overhead of the compiled code will fall between the two measured extremes depending on how often parallel tasks are used. Additionally, our proof of concept implementation does not exploit possible optimizations which can reduce the overhead of the translation.

IX. Summary and Conclusion

In this paper we focus on business processes using wireless sensor networks. Our main contribution is enabling business analysts to control the execution of such a process on the abstract level of its BPMN model. Furthermore, we enable them to test and debug their models on the same level of abstraction. By this means, we close the gap between the specification and the actual implementation of business processes.

We achieve this by introducing a compiler which translates the sequential execution semantics of BPMN models into event-driven code suitable for common sensor network platforms. Software developers provide a set of building blocks which can be reused by different models. These building blocks can be either directly programmed or also modelled in BPMN depending on the desired level of abstraction.

We additionally elaborated on the set of required tools and presented a proof of concept implementation thereof. Our evaluation results on several case studies show that the generated code is only 1% less energy efficient than handwritten equivalents. Concerning RAM and flash usage, though, the overhead is higher. Nevertheless, we could show that the generated applications fit on standard motes.

In summary, we argue that the introduced overhead is worth the benefits of keeping business processes synchronized with their implementations. We argue that the parts of a business process which involve WSN can and should be automatically generated directly from the model specification. Thus, organizations can dynamically adapt their technology implementation according to their business processes and not vice versa.

References

[1] Spieß, P. et al., "Integrating Sensor Networks with Business Processes," in *Work. on Real-World Sensor Networks at ACM MobiSys*, 2006.

[2] S. Schäfer, "Secure Trade Lane: A Sensor Network Solution for More Predictable and More Secure Container Shipments," in *Comp. to the 21st Symp. on Object-Oriented Programming Systems, Languages, and Applications*, 2006, pp. 839–845.

[3] S. Krishnamurthy and L. Lange, "Distributed Interactions with Wireless Sensors Using TinySIP for Hospital Automation," in *Proc. of the 6th Conf. on Pervasive Computing and Communications*, 2008, pp. 269–275.

[4] M. zur Muehlen and D. T. Ho, "Service Process Innovation: A Case Study of BPMN in Practice," in *Proc. of the 41st Conf. on System Sciences*, 2008, pp. 372–372.

[5] F. P. Brooks, Jr., *The Mythical Man-Month*. Addison-Wesley, 1995.

[6] OMG, "Business Process Model and Notation," www.bpmn.org.

[7] A. Reinhardt and R. Steinmetz, "Exploiting Platform Heterogeneity in Wireless Sensor Networks for Cooperative Data Processing," in *Proc. of the 8th GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze"*, 2009, pp. 21–24.

[8] OMG, "Unified Modeling Language," www.uml.org.

[9] IBM, "Rational Rhapsody," www.ibm.com/software/awdtools/rhapsody.

[10] G. Fuchs and R. German, "UML2 activity diagram based programming of wireless sensor networks," in *Proc. of the ICSE Work. on Software Engineering for Sensor Network Applications*, 2010, pp. 8–13.

[11] Sun Microsystems, Inc., "Sun SPOT World," www.sunspotworld.com.

[12] M. Ghercioiu, "A Graphical Programming Approach to Wireless Sensor Network Nodes," in *Proc. of the Sensors for Industry Conf.*, 2005.

[13] M. M. R. Mozumdar et al., "A Framework for Modeling, Simulation and Automatic Code Generation of Sensor Network Application," in *Proc. of the 5th Conf. on Sensor, Mesh, and Ad Hoc Communications and Networks*, 2008.

[14] National Instruments, "LabVIEW," www.ni.com/labview.

[15] Mathworks, "Simulink," www.mathworks.com/products/simulink.

[16] Cheong, E. et al., "Viptos: A Graphical Development and Simulation Environment for TinyOS-based Wireless Sensor Networks," University of California at Berkeley, Tech. Rep. UCB/EECS-2006-15, 2006.

[17] T. Naumowicz et al., "Prototyping a Software Factory for Wireless Sensor Networks," in *Proc. of the 7th Conf. on Embedded Networked Sensor Systems*, 2009, pp. 369–370.

[18] A. Pathak et al., "Srijan: A Graphical Toolkit for Sensor Network Macroprogramming," in *Proc. of the the 7th Joint Meeting of the European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*, 2009, pp. 301–302.

[19] OASIS Technical Committee, "Web Services Business Process Execution Language Version 2.0." www.oasis-open.org/committees/wsbpel.

[20] N. Glombitza et al., "Using Graphical Process Modeling for Realizing SOA Programming Paradigms in Sensor Networks," in *Proc. of the 6th Conf. on Wireless On-Demand Network Systems and Services*, 2009, pp. 57–64.

[21] ——, "Integrating Wireless Sensor Networks into Web Service-based Business Processes," in *Proc. of the 4th Work. on Middleware Tools, Services and Run-Time Support for Sensor Networks*, 2009, pp. 25–30.

[22] Spieß, P. et al., "SOA-Based Integration of the Internet of Things in Enterprise Services," in *Proc. of the Conf. on Web Services*, 2009, pp. 968–975.

[23] C. Favre and H. Völzer, "Symbolic execution of acyclic workflow graphs," in *Proc. of the 8th Int. Conf. on Business Process Management*, 2010, pp. 260–275.

[24] A. Dunkels et al., "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Proc. of the 29th Conf. on Local Computer Networks*, 2004, pp. 455–462.

[25] J. Hill et al., "System Architecture Directions for Networked Sensors," *SIGPLAN Notices*, vol. 35, no. 11, pp. 93–104, 2000.

[26] A. Caracaş et al., "Mote Runner: A Multi-language Virtual Machine for Small Embedded Devices," in *Proc. of the 3rd Conf. on Sensor Technologies and Applications*, 2009, pp. 117–125.

[27] L. Motolla and G. Picco, "Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art," *ACM Computing Surveys*, to appear.

[28] Oryx, "Oryx Editor," bpt.hpi.uni-potsdam.de/Oryx.

[29] L.S. Bai et al., "Archetype-based design: Sensor network programming for application experts, not just programming experts," in *Proc. of the International Conference on Information Processing in Sensor Networks*, 2009, pp. 85–96.

[30] Memsic, "IRIS Wireless Measurement System," www.memsic.com.

Some of the product, company or services names referred to in this paper may be trademarks or registered trademarks of third parties in the USA, other countries, or both. All web references were last accessed in February 2011.