Model-Driven Construction of Embedded Applications Based on Reusable Building Blocks – An Example

Frank Alexander Kraemer, Vidar Slåtten, and Peter Herrmann

Norwegian University of Science and Technology (NTNU),
Department of Telematics, N-7491 Trondheim, Norway
{kraemer,herrmann,vidarsl}@item.ntnu.no

Abstract. For the rapid engineering of reactive systems we developed the SPACE method, in which specifications can be composed of reusable building blocks from domain-specific libraries. Due to the mathematical rigor and completeness with which the building blocks are designed, we can provide tool support facilitating a high degree of automation in the development process. In this paper, we focus on the design of embedded Java applications executed on Sun SPOTs by providing dedicated blocks to access platform-specific functionality. These building blocks can be used in combination with other blocks realizing protocols such as leader election to build more comprehensive applications. We present an example specification and discuss its automatic verification, transformation and implementation.

1 Introduction

Maybe it is just that engineers still love the LEGO bricks of their childhood, but creating software systems by connecting reusable building blocks seems to be an attractive development paradigm that can facilitate reuse and enable an incremental development style in which problems can be solved block by block. Yet the everyday practice by developers often does not work as smoothly as simply plugging together bricks: Major challenges lie in the nature of reusable modules in the first place, especially in how to encapsulate and how to compose them. Our engineering method SPACE [1,2] aims to address these issues. As reusable units we use special building blocks that express their behavior in terms of UML activities. These can be composed by pins, and a system can be constructed as a hierarchy of building blocks. While building blocks can describe local behavior executed by a single component, they can in general also cover *collaborative* behavior among *several* components. This facilitates the reuse of solutions to problems that require the coordination of several components, and is especially useful to describe services.

While our method is general and useful in a variety of domains, we demonstrate in this article its application in the area of embedded systems. For that, we present the results of a case study on a sensor network carried out as part

R. Reed, A. Bilgic, and R. Gotzhein (Eds.): SDL 2009, LNCS 5719, pp. 1–18, 2009.

of the applied research project ISIS[1] (Infrastructure for Integrated Services [3]), in which we develop methods, platforms and tools for the model-driven development of reactive systems for applications in home network systems. The case study is implemented on small processing devices from Sun Microsystems, called *Sun SPOTs* [4] that run Java.

In the following, we cover all steps needed to realize deployable code from high-level specifications. We will focus especially on the definition of building blocks for the domain of Sun SPOTs and on a protocol for fault-tolerant leader election. We start with an introduction of Sun SPOTs including the runtime support system, followed by a brief overview of our method. In Sect. 2, we present the example system and its high-level specification based on UML activities. The next two sections document our library for Sun SPOTs and the leader election algorithm. In Sect. 5 and 6, explanations of the automated analysis and implementation follow, in which state machines similar to SDL processes are synthesized, from which code is generated.

1.1 Embedded Java on Sun SPOTs

A sketch of a Sun SPOT is shown on the left side of Fig. 1. Each SPOT is equipped with two buttons and sensors for temperature, light and acceleration. SPOTs can also carry extension cards to interact with various other devices. A Sun SPOT is controlled by a 32-bit ARM 9 processor that can run the Java virtual machine *Squawk* [5] executing Java 1.3 code following the CLDC 1.1 specification. SPOTs can communicate among each other using IEEE 802.15.4 radio communication, and build a mobile ad hoc network.

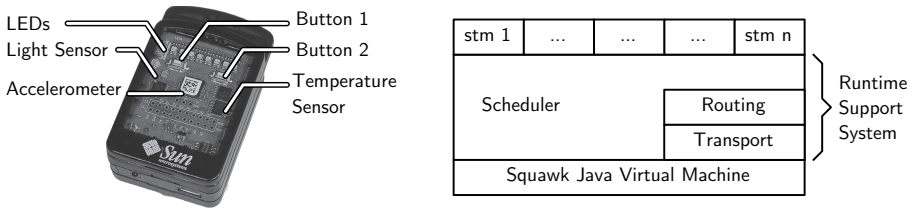

Fig. 1. Sun SPOT and Runtime Support System

1.2 Runtime Support System

To facilitate the execution of many concurrent processes on Sun SPOTs, we have implemented a runtime support system [6], sketched on the right side of Fig. 1. It includes a scheduler that is responsible for triggering the execution of state machine transitions whenever signals are received or timers expire. Further, a router and an object responsible for the transport of signals support communication using the SPOT's radio communication. For a detailed description of

[1] Partially funded by the Research Council of Norway, project #180122.

the execution mechanisms and their formal behavior in temporal logic, we refer to [7]. To generate the state machine classes from UML state machines, we use the code generator described in [8,9], which produces the necessary Java code.

1.3 The SPACE Engineering Method

We developed the method SPACE [1,2] for the engineering of reactive systems. This method focuses on the definition of reusable building blocks expressed as UML activities and collaborations, combined with Java code for detailed operations. Building blocks are grouped into libraries for specific domains, as illustrated on the left hand side of Fig. 2. Developers can use these blocks by composing them together within UML collaborations and activities: the collaborations describe the structural binding of roles and provide a high-level overview and activities describe the detailed behavioral composition of events, with some additional glue logic where necessary. Each block has an associated *external* state machine, abbreviated ESM, that provides a behavioral contract describing in which sequence parameters must be provided to or may be emitted by a block. This description is useful for understanding a block without looking at its internal details, and enables compositional model checking, as we describe below.

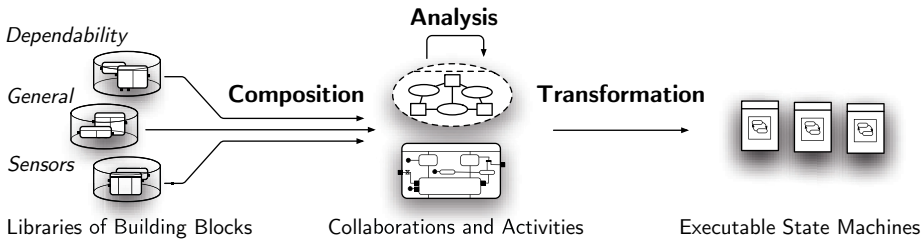

Fig. 2. The SPACE engineering method

Once a specification is complete, it is analyzed to ensure various properties that should hold for any application. For example, a composition of blocks should never harm any of the contracts (ESMs) and a collaboration should terminate consistently. For this behavioral analysis, we use model checking. Due to the compositional semantics and the encapsulation of building blocks by their ESMs, the state space needed for model checking tends to be very small, since only one building block on a single decomposition level has to be considered at a time.[2]

Complete systems are represented by special system collaborations and activities. When a system is sound, it can be transformed automatically into executable state machines and components, using a model transformation [10,11]. From the resulting state machines, code for different platforms (such as the Sun SPOTs introduced above) can be generated.

[2] We observe that most building blocks in our libraries require far less than 100 states.

2 A Sensor Network for Remote Home Monitoring

An increasingly popular area for home automation is to remotely monitor vacation homes and cabins. Several sensors can be installed in a cabin. One of the assumptions in our project is that embedded sensors with processing capacity similar to Sun SPOTs are so cheap they can also be used in a consumer market. For instance, the sensors can register the temperature at several places, detecting frost or fire. Further, they can detect sudden changes in light or measure acceleration on doors and windows, indicating that somebody is breaking in. With the extension card presented in [12], we further assume that each Sun SPOT is capable of GSM communication to set off an alarm to a remote user, for example by means of an SMS.

To improve the quality and robustness of the system, the sensors communicate among each other before sending an alarm via GSM. This serves several purposes: First, multiple sensors can be used redundantly, so that important conditions are monitored by more than one sensor, whereas only one alarm should be issued. Second, some conditions may give rise to alarm if the sensors are triggered in a certain pattern. For example, while changes in light of one sensor could indicate a broken window shutter, a change observed by several sensors may simply be due to a cloud moving in front of the sun.[3] This means that alarms need to be coordinated. For that reason, we use a leader election protocol that points out one SPOT sensor to filter and issue alarms. If the leader runs out of battery or otherwise fails, a new leader takes over. Such a network is illustrated in Fig. 3.

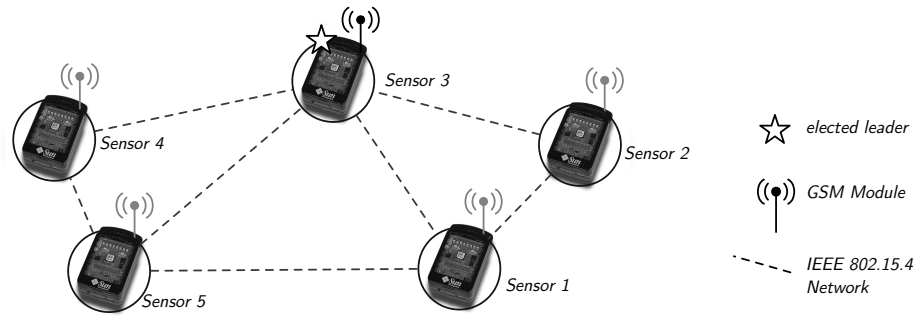

Fig. 3. Sensor network with the elected leader

Figure 4 shows the UML activity describing the behavior of a SPOT sensor as composed from our reusable building blocks. Since the SPOT sensors of the system all have the same behavior, it suffices to specify only one of them. To visualize the relationship of a SPOT sensor to the other sensors explicitly, however, we use two activity partitions. The left one, *spot sensor*, describes how a SPOT sensor is composed from building blocks, which defines the behavior. The

[3] We will not discuss detailed patterns describing when an alarm should be triggered, and we will also disregard the configuration of individual SPOT sensors.

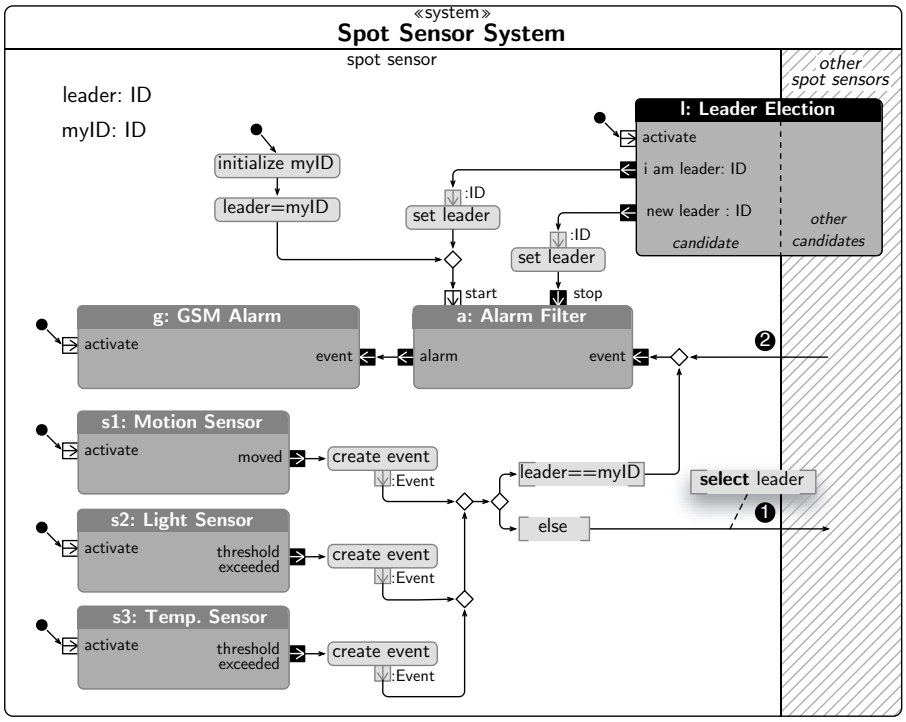

Fig. 4. Activity describing the composition of SPOT sensors from building blocks

right partition, *other spot sensors*, enables us to represent the communication with the other sensors. This partition is only sketched, as only the left one will be used for the transformation and code generation.

A sensor consists of a block[4] for GSM communication g, the alarm filter a and three building blocks accessing the Sun SPOT's sensors for motion ($s1$) light ($s2$) and temperature ($s3$). While these blocks encapsulate local behavior, a building block can also comprise collaborative behavior that is executed by several participants. The leader election, contributed by building block l in Fig. 4, is a typical example for that. It is a collaboration among several SPOT sensors, and therefore crosses the activity partitions. Internally, the block specifies the establishment of contact between all the sensors and how a leader is selected amongst them. This behavior is detailed in Sect. 4.

The activity also contains references to the operation *create event*. Since UML does not have a concrete language for actions, the details of these operations are specified by Java methods, managed by our editor. The other elements in the activity are initial nodes (●) as well as merge and decision nodes (◇). Decision nodes are followed by flows that are guarded ([ ]).

[4] Technically, blocks are modeled as UML elements of type *Call Behavior Action*, which can refer to subordinate activities.

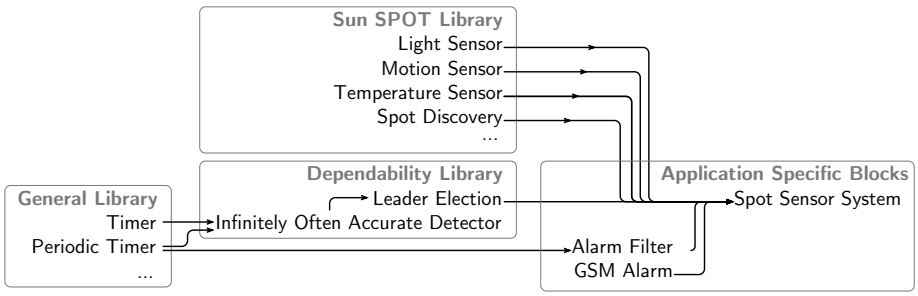

Fig. 5. Overview of reused block from libraries and application-specific blocks

Upon the start of a SPOT sensor, the initial nodes emit a token and start all blocks, including the collaboration for the leader election. The alarm filter is started as well, so that the SPOT by default uses its own GSM Alarm block to send any SMS notifications, until it finds another leader. The leader election emits a token through *new leader* once it detects a SPOT that is pointed out as the new leader, carrying its ID. In case a SPOT itself is pointed out as leader, a token is emitted through *i am leader*. In both cases, the ID of the leader is stored in variable *leader*. If a SPOT becomes leader, the alarm filter is started, and if the SPOT loses its leader status, the alarm filter is terminated.

Whenever one of the sensors *s1*, *s2* or *s3* registers a condition, it emits a token via its output pin, upon which an event is created containing the kind of condition and ID of the sensor. If the SPOT owning the sensors has the leader role (i.e., guard *leader==myID* is valid), the event is directly passed to the alarm filter. Otherwise, the SPOT sensor forwards the event to the current leader. In this case, the leader is one of the other SPOT sensors, and sending to it is specified by the transfer edge ❶. Since the other SPOTs are potentially many, we have to select which one to address, using the **select** operator introduced in [10]. It refers to the ID of the leader. Vice versa, if a SPOT sensor has the leader role, it may receive events from other SPOT sensors (at ❷).

Figure 5 provides an overview of the dependencies between the building blocks used for the specification of the SPOT sensor system. Most of them are taken from our existing libraries (listed here with only those blocks used in the example). The Alarm Filter, the experimental GSM Alarm, and the complete system are specific for the example.

3 Building Blocks Specific for Sun SPOTs

Our library for Sun SPOTs contains twelve building blocks dedicated to the specific capabilities of the devices, such as the buttons, all sensors on the SPOTs, and the LEDs. In the following we present some of those that are used in the SPOT sensor system.

3.1 Building Block for Sensors

Figure 6 shows the internal details of the block for the detection of movements. The accelerometers of the Sun SPOTs are accessible via a special API. To react on sudden accelerations that exceed a certain threshold value, a listener is registered at the SPOT classes that provide access to the hardware. To keep the execution of the code reacting upon an event under the control of the scheduler of our runtime support system (RTS), the building block uses an internal signal as buffer, to decouple the processes. For this reason, operation *register listener* creates a listener, which, upon its invocation following a sudden movement, produces a signal *MOVED*, that is fed into the RTS. Once this signal is processed, the behavior following the accept signal action declared for *MOVED* in Fig. 6 is executed: a token is emitted via output node *moved*, and the listener is reactivated, to listen for further movements. The blocks controlling the light and temperature sensors access the SPOT API in a similar way.

On the right hand side of Fig. 6, the ESM for the motion sensor is shown. As mentioned previously, it documents the behavior visible at the pins of an activity, so that we know its external behavior when it is instantiated as a block as in Fig. 4. Due to the ESM, we know that after a token enters *activate*, tokens may be emitted via *moved* until we terminate the block via *stop*.

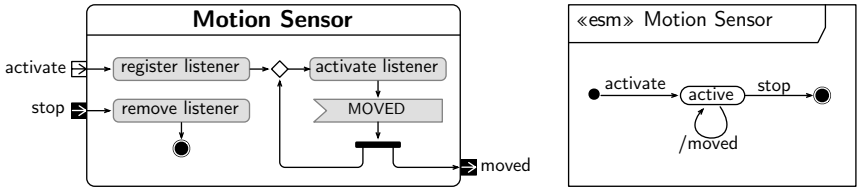

Fig. 6. Building block for the motion sensor

3.2 SPOT Discovery

To dynamically find other SPOTs in the sensor network, we provide a collaborative building block which uses the Sun SPOT's broadcasting functions so that they can discover each other. The corresponding activity is shown in Fig. 7. The partition *beacon* describes how a SPOT that wants to be discovered sends out periodic messages. Since these messages are specific for Sun SPOTS, they are sent directly from the Java operation, instead of using our runtime support system. The partition *listener* describes the logic to be implemented by a Sun SPOT that wants to discover other SPOTs. For that, it listens to the incoming beacon messages. To decouple the receiving processes from the scheduling of state machine transitions, once such a message arrives, it is fed into our RTS via signal *FOUND*, similar to the listener reacting to the movement of a SPOT explained above. If the ID is not yet known, a token is emitted via *found spot*. Notice that if a SPOT wants to both discover other SPOTS and be discovered, it instantiates this collaboration twice, once as a beacon and once as a listener.

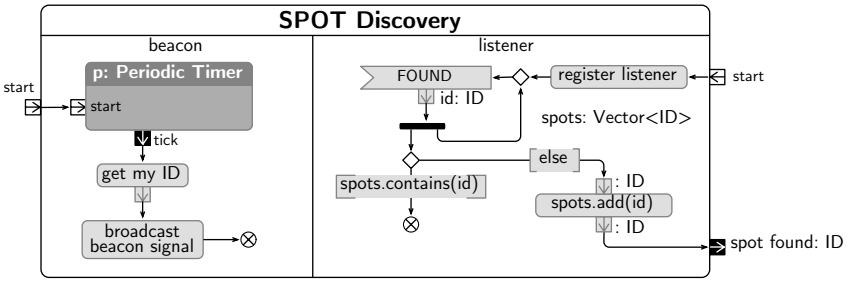

Fig. 7. Building block for the service discovery

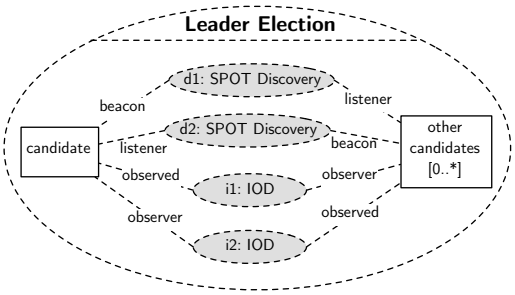

Fig. 8. Collaboration for the leader election

4 Collaborative Building Blocks for Leader Election

To make sure that only one of the SPOT sensors forwards an alarm over GSM, we use a fault-tolerant leader election protocol. Should the leader SPOT run out of battery or otherwise fail, another one must take its place so that alarms are still sent if necessary. To solve this problem, we implemented an algorithm from [13]. The algorithm uses an *Infinitely Often Accurate Detector* (IOD) as failure detector, a concept from [14], which is used by a component to monitor if any of its communication partners have crashed.[5] In Sect. 4.1 we provide a dedicated building block for this function.

The collaboration in Fig. 8 specifies the structural aspects of the leader election. It depicts the participant *candidate* as collaboration role, and refers to the sub-services for SPOT discovery and failure detection by collaboration uses *d1, d2* and *i1, i2*. The leader election is a symmetric collaboration, in which all participating roles have the same behavior, and the role for the candidate is therefore represented twice. For the model transformation and the code generation, the left *candidate* is used. To make the collaboration with the other candidates explicit, we refer to the *other candidates* on the right hand side, similar to our proceedings with the SPOT sensors in Sect. 2.

[5] In the fault-tolerance domain, a node is said to *crash* if it from some point on permanently ceases all operations, but works correctly until then (see [15]).

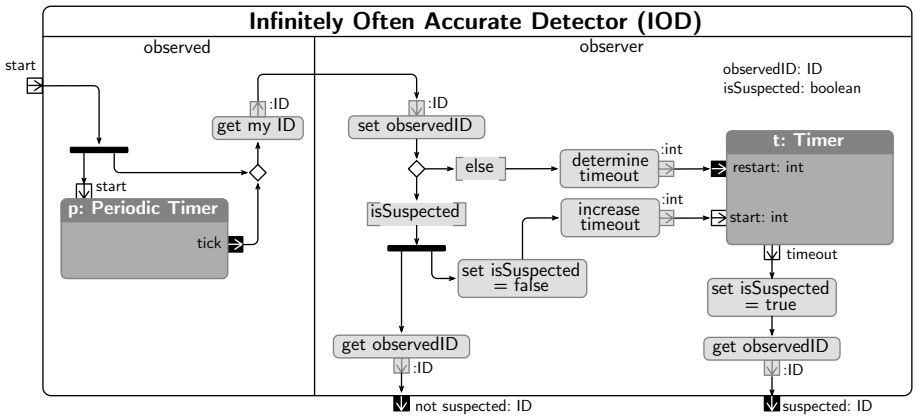

Fig. 9. Building block for the Infinitely Often Accurate Detector

4.1 Infinitely Often Accurate Detector (IOD)

In our example, we use the *Infinitely Often Accurate Detector* (IOD, [13]) as specified in Fig. 9. The partition on the left side models the observed SPOT, which periodically sends so-called "alive" messages to the observing SPOT, represented by partition *observer*. These messages are triggered by the periodic timer p and carry the ID of the observed SPOT. The observer SPOT maintains two variables to store the status of the observed SPOT; *observedID* for its ID and the boolean *isSuspected*. Moreover, the observer has a timer t to determine if the alive message from the observed SPOT is delayed.

Whenever the observer receives an alive message from the observed SPOT, it reacts depending on the current value of *isSuspected*:

- If the observer does not suspect the observed SPOT sensor of having crashed, it will simply restart timer t and wait for the next alive message.
- If, on the other hand, the observer currently suspects the observed SPOT of having crashed, the observer will change *isSuspected*, increment the timeout period[6] and emit the observed's ID through output node *not suspected*.

If, however, timer t expires (i.e., no alive message was received in time), the observer will suspect the observed SPOT of having crashed, set *isSuspected* accordingly and emit a token carrying the observed SPOT's ID through output node *suspected*.

Since a message could also be delayed in the communication medium, a timeout does not always mean that a SPOT has crashed. Hence there may exist transient states in which two SPOTs are both considered the leader. This, however, is acceptable for our application domain. For a detailed analysis and proof of the properties of the Infinitely Often Accurate Detector, we refer to [13].

[6] Incrementing the timeout period upon detecting a false suspicion ensures that the observer will wrongly suspect the observed only a limited number of times.

4.2 Composed Building Block for the Leader Election

The detailed behavior of the leader election is expressed by the activity in Fig. 10. Similar to the overall system of Fig. 4, the leader election is symmetric. The partition *candidate* on the left side represents one participant and its detailed behavior, while the partition to the right represents its communication partners.

As part of the leader election, a SPOT participates in the Infinitely Often Accurate Detector (IOD) collaboration as both *observer* and *observed* entity. This is represented by blocks *i1* and *i2*, which both refer to the activity in Fig. 9, but which are bound to partition *candidate* with roles *observed* resp. *observer*. Moreover, this collaboration is executed as multiple concurrent sessions (once towards each communication partner). This is signified by the shadow around them, a notation introduced in [10].

When the leader election collaboration is activated, the *SPOT Discovery* collaboration is initialized as both beacon (*d1*) and listener (*d2*), according to the role binding in Fig. 8, so that a SPOT sensor can both detect others and be detected by others. For each sensor found, a token with its ID is emitted via pin *spot found* of *d2*. This ID is used to start a new session of the IOD collaboration *i1*, so that a SPOT is observed by any other SPOT it detects. For that we use again the **select** statement, which this time refers to the value provided by the token flow. Vice versa, once a SPOT is detected by other SPOTS, they start a new instance of the IOD collaboration (in this direction represented by *i2*).

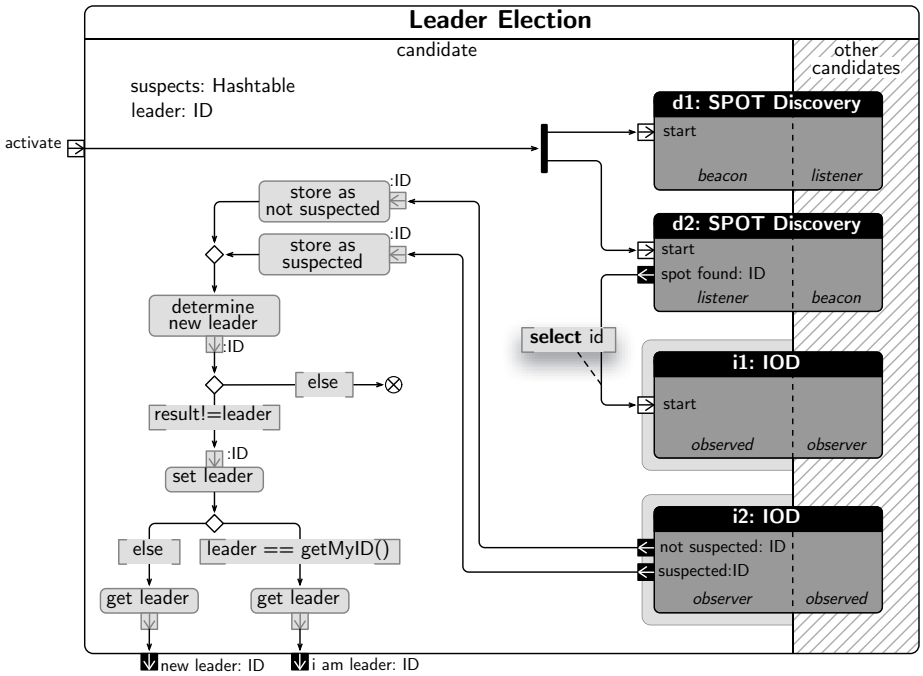

Fig. 10. Building block for leader election

Via the output pins *suspected* and *not suspected* on *i2*, a SPOT is notified about perceived changes in the state of each of the other SPOTs. The logic that follows determines the current leader status. For that, hash table *suspects* maps the ID of the other SPOTs to their respective status (suspected or not suspected). Whenever *i2* issues a change in state of another SPOT via one of its output pins, the subsequent operations store this change to the hash table and determine the new leader. If several SPOTs qualify for the leader status, the one with the lowest ID is chosen. If the leader has changed, we store the new leader and check if the new leader is this SPOT. Depending on the outcome, a token is emitted through either the *i am leader* or *new leader* output node.

5 Automated Analysis

The analysis of the specification is based on model checking. This process is automated, since our tool also generates the corresponding theorems to be verified. Currently, we check the following generally desirable system properties [16]:

- A building block must conform with its own ESM. The motion sensor of Fig. 6, for instance, may not emit a token via node *moved* after the surrounding context provided one via *stop*.
- A building block must also obey all ESMs of the subordinate blocks it is composed from.
- Building blocks with more than one participant are checked for bounded communication queues. For the IO detector in Fig. 9, for instance, we find that the periodic timer could, in principle, overflow the queue between the observing and the observed component.[7]

The analysis focuses on the soundness of interactions among collaboration participants as well as the correct composition of all building blocks with respect to event orderings. The content of operations (that is, the Java code) is not part of the analysis. In cases where decisions are involved that depend on variables, the analysis always examines all alternative branches. If the executions of some branches may harm certain properties, we reason manually if these cases may in fact happen. For instance, in the IO detector of Fig. 9, the else branch may restart the timer before it is started. This, however, never happens in the final system because of the value of *isSuspected*.

The results of the analysis are presented to the user by explanatory annotations within the original UML model, so that no expertise in the underlying formalism is required, as demonstrated in [17]. In addition, counter examples illustrating design flaws are presented as animations within the activities. In our experience, checking the above mentioned properties is of great value in the practical development of specifications. Although these properties may appear simple when considered in isolation, even experienced engineers usually harm

[7] In this case, however, we estimate the time needed for the transmission and subsequent processing and conclude that this is not an issue in a real system.

several of them in initial designs, especially when more complex collaborations are constructed.

Due to the compositional semantics of our method, each building block can be analyzed separately. Internal building blocks are abstracted by their ESMs, so that the global state space of the specification in Fig. 4 has only 15 distinct reachable states. Moreover, since most of the building blocks are taken from libraries and are already analyzed, only the new ones created for the specific applications have to be examined. These are the ones for the *SPOT Sensor System*, the *Alarm Filter* and the *GSM Alarm*.

6 Automated Implementation

As briefly mentioned in the introduction, the implementation is performed by a completely automated process with two steps: In a first step, executable state machines are synthesized from the activities. In a second step, code is generated. This is possible since the activities provide descriptions that are behaviorally complete, and the details of operations are provided as Java methods as part of the building blocks.

6.1 Transformation to Executable State Machines

In Fig. 11 and 12, we present the state machines as generated by the transformation. In our method, they are only an intermediate result used as input for the subsequent code generation; developers do not have to edit or read them. In the following, we highlight some properties to demonstrate the soundness of the transformation.

For the partitioning of components into state machines (or *processes* in SDL), our algorithm follows the guidelines from [6]. In particular, the algorithm merges all behavior of building blocks that is executed one at a time by the component under construction into one single state machine. All blocks that denote multi-session collaborations (behavior that is executed multiple times towards a changing number of different communication partners) are implemented by dedicated state machines, one instance for each session, as presented in [10]. For the SPOT sensor system, for instance, the algorithm creates the state machine *Spot Sensor*, depicted in Fig. 11, which takes care of the main component behavior. This includes all logic contained in the building blocks used in Fig. 4. However, since the behavior of the Infinite Often Accurate Detector is executed concurrently within each SPOT sensor (once for each other sensor detected), its behavior is implemented by dedicated state machines. These are state machines *Observer* and *Observed* in Fig. 12.

The main state machine *Spot Sensor* has two[8] distinct control states, *1* and *2*. This is because the transition behavior only has to distinguish if a SPOT is

[8] This is less than the 15 states from the previous analysis because the analysis also captures the interleaving with other SPOTs and the queues for communication, which do not contribute any control states for a local component.

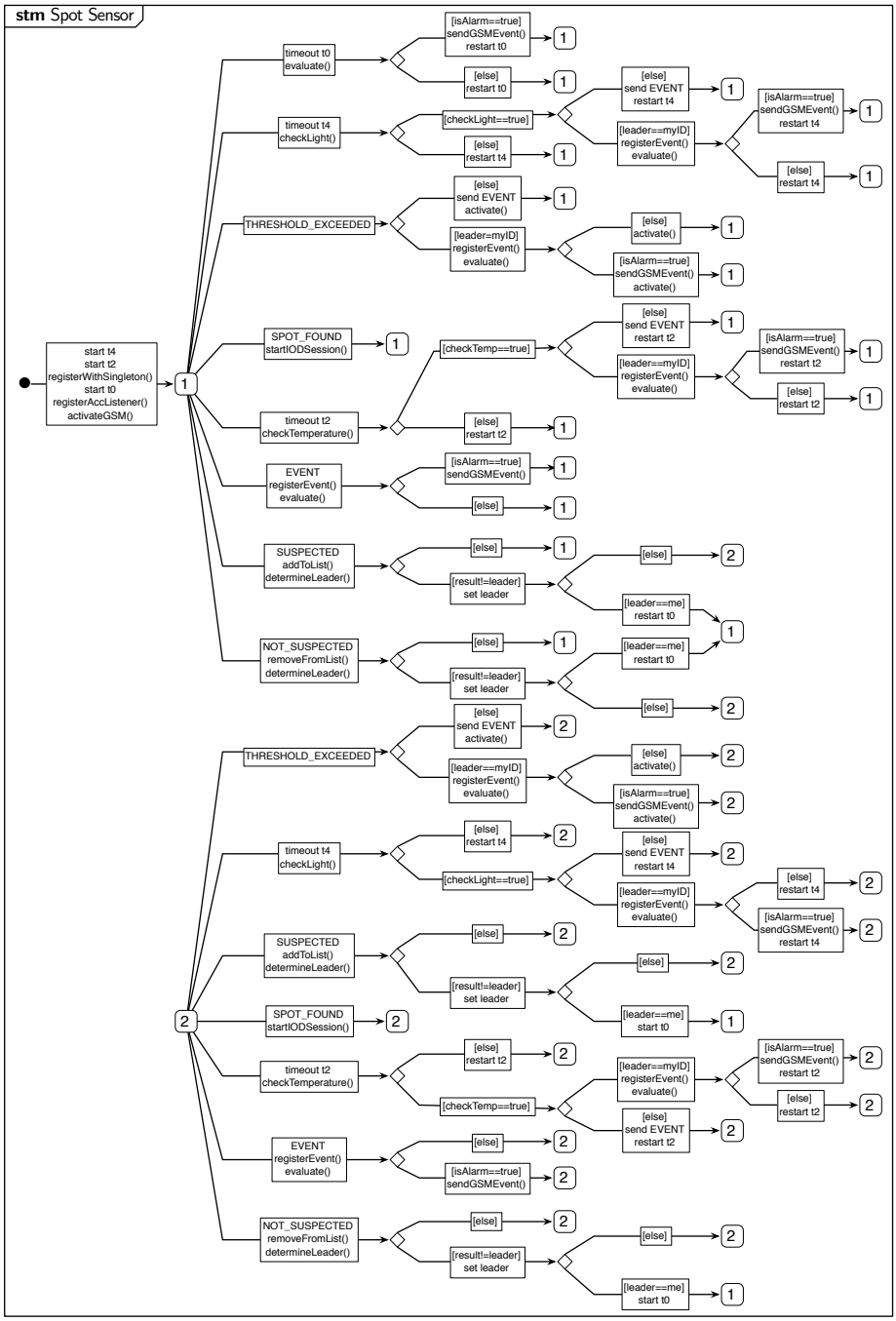

Fig. 11. Bird's eye view of the synthesized state machine for the Spot Sensor

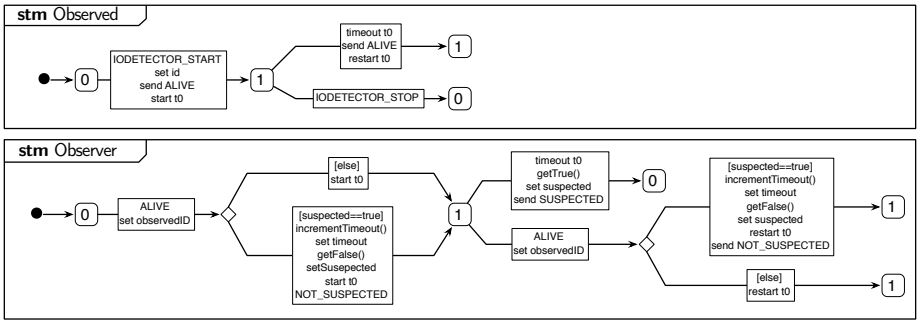

Fig. 12. The synthesized state machines for the IO detector

the leader or not. When a spot is the leader, the alarm filter is active and the state machine is in state *1*. When another SPOT is the leader, the alarm filter is inactive and the state machine is in state *2*. The transitions from either state handle the periodic checks of the sensors, the periodic discovery protocol and react to the events of the Infinitely Often Accurate Detector. In state *1*, which is entered by the initial transition, the SPOT assumes it is the leader and therefore starts the alarm filter, which constantly evaluates the log of events, shown by the topmost transition.

6.2 Code Generation for Sun SPOTs

Since the Sun SPOTs execute Java, the code generator described in [9] is largely based on the standard Java code generator, described in [18]. As introduced in Sect. 1.2, the execution is based on a runtime support system, which takes care of scheduling, routing and transport of messages. The scheduler (see Fig. 1) maintains event queues for each state machine in which incoming messages and active timers are placed. In a round-robin manner, the scheduler triggers the execution of state machine transitions by feeding the event into a dedicated transition method, which is specific for each state machine type. The transition method contains nested if-statements that distinguish the current control state and input event and then execute the effect as specified by the UML transitions in Fig. 11 and 12. Effects referring to operation calls on the activity level, such as *determine new leader* in Fig. 10, are copied into the transition method. Other actions that are part of a transition effect, such as sending signals or operations on timers, are synthesized from the UML model. The transport module (see Fig. 1), responsible for sending and receiving messages from and to other SPOTs, uses the the radio stream protocol from the Sun SPOT API to transmit messages. This protocol provides buffered, reliable, stream-based communications over multiple hops on top of the IEEE 802.15.4 radio protocol. The content of the messages sent via the radio channels are SOAP-documents generated with the help of the kSOAP libraries [19], as described in [8]. For the necessary serialization of objects, the code generator adds methods that convert objects and primitive types to strings.

7 Estimation of Reuse Proportions

To estimate the degree of reuse for the exemplified system, we distinguish between the building blocks that are part of our libraries and intended for reuse, and those building blocks constructed specifically for the application. These are shown in Fig. 5, with the libraries on the left hand side. As application-specific we count the *Alarm Filter*, the *GSM Alarm* and the overall *SPOT sensor* system. The effort necessary for the construction of a building block consists of the UML models on the one hand and Java code contained within the call operation actions (like *determine new leader* in Fig. 10) on the other hand.

- By counting the lines of code contained in the call operation actions in each building block, we find that there are $l_{blocks} = 443$ lines of code within the call operation actions for all building blocks used in the system in total. Those building blocks taken from libraries contribute with $l_{lib} = 333$ lines, so that the reuse proportion $R_{code} = l_{lib}/l_{blocks}$ is 75 %.
- As an estimate for the effort spent UML modeling, we use a simple metric that just counts the number of activity nodes and activity edges $n = n_{nodes} + n_{edges}$ within a building block. This metric shows that all building blocks used in the system consist of $n = 276$ edges and nodes in total. Those building blocks taken from the library contribute with $n_{lib} = 195$ elements, so that the reuse proportion $R_{model} = (n_{lib}/n)$ is 71 %.

Of course, these numbers vary for different systems. For the given example, we have programmed a relatively simple logic for the alarm filter, which contributes only 50 lines of code. Since the GSM module is not yet finalized, we estimate another 50 lines for that building block.

To get an impression of the overall gains including the automatic implementation, we consider also the complete code needed for the execution on top of the runtime support system. The code generated automatically for the state machine logic adds up to $l_{stm} = 634$ lines, and the number of code lines written manually for the Java operations copied from the building blocks as mentioned above is $l_{blocks} = 443$. This means that the code necessary for the entire application has $l_{total} = l_{stm} + l_{blocks} = 1077$ lines,[9] from which $l_{stm}/l_{total} = 59$ % are generated automatically. If we add up these numbers, we find that $(l_{lib} + l_{stm})/l_{total} = 90$ % of the Java code lines are either reused or generated from the UML models.

8 Related Work

There exist a number of approaches for the model-based design of reactive systems that are also suitable for embedded applications. Some of them based on SDL such as TIMe [20], SPECS [21], SOMT [22] and SDL-MDD [23]. Others,

[9] The underlying runtime support system has about 1900 lines of code. Since it is provided as a library that can be reused also in manual approaches, it is not part of our calculation.

such as ROOM [24] (later UML-RT) or Catalysis [25], are oriented towards UML as language. As design models that describe the behavior of individual components, these approaches use state machines, either in the form of SDL processes or as UML state charts (called *ROOM charts* in [24]). To capture collaborative behavior among several components, most of these approaches rely on MSCs. Catalysis [25], inspired by the Object-Oriented Role Analysis Method (OOram, [26]) and DisCo [27], on the other hand, uses collaborations more explicitly in specific diagrams, albeit in a rather informal way that requires manual synchronization by the developers. Micro protocols [28] are another approach to capture and encapsulate communication protocols within self-contained units, by using pairs of SDL processes or composite states.

In principle, these approaches are compatible with the one presented here, since all the design models based on state machines with their emphasis on event-driven transitions are quite similar. The difference lies in the models on which developers work: To enable the composition of collaborative behavior as self-contained building blocks, we use UML activities, from which the state machine-based design models are derived automatically. This enables a number of opportunities for the reusability, the analysis and the overall specification style, as we will argue below.

9 Concluding Remarks

In our experience, the composition as enabled by activities, shown for example in Fig. 4, is quite flexible. We attribute this to two major reasons: First, the complete but cross-cutting nature of UML activities, in which the coordination of several participants can be described within the same diagram. If, for example, we would like to exchange the selected leader election protocol with another one, we would just have to replace the building block l in Fig. 4, and its connections to the other blocks, which can be achieved by focusing on one single diagram. Second, the way activities enable the encapsulation of functionality related to a certain purpose as separate, self-contained building blocks. While state machines offer some means of structuring (for example composite states), they do not offer the same degree of flexibility and separation as activities. The functions encapsulated by the building blocks in Fig. 4, for example, are dispersed among several transitions in the state machines of Fig. 11 and 12. One reason for that is that state machines represent their states by explicit control states, while activities use concurrent flows that may execute independently. Although such behavior can to a certain degree be described in state machines by concurrent regions, such a description style gets intricate once the behaviors in these regions need to be synchronized. However, since state machines are very suitable for the specification of the executable behavior of components, we generate them in the described way, so that we have both the compositional features of UML activities and the efficient scheduling of state machines.

Besides these properties coming from the chosen notation, an important feature of our method is the compositional verification it enables, based on the

underlying semantics in cTLA [29]. Not only does this reduce the state space during model checking, but it also has important effects on the larger scale development process. Since building blocks can be verified individually, proven solutions can be encapsulated in building blocks, and these can be checked and stored in a library. Whenever a building block is reused, the verified properties are enforced automatically and do not have to be re-verified. This enables "true reuse" as mentioned in [25], in which reuse does not mean to simply copy and paste some parts of a specification, but also ensures that important properties are maintained.

All things considered, we think that the chosen principles and the way they are combined enable a reuse-oriented specification style, one that encourages the use of encapsulated building blocks to a high degree, but that still allows us to adapt systems to match the requirements of the individual application. This is a crucial step towards the cost-effective LEGO-brick like development paradigm.

Acknowledgements. We would like to thank Fritjof Boger Engelhardtsen from Telenor for support with the GSM module, as well as Marius Bjerke and Bemnet Tesfaye Merha for their enthusiastic work on the Sun SPOTs in their theses.

References

1. Kraemer, F.A.: Engineering Reactive Systems: A Compositional and Model-Driven Method Based on Collaborative Building Blocks. PhD thesis, Norwegian University of Science and Technology, Trondheim (August 2008)
2. Kraemer, F.A., Herrmann, P.: Service Specification by Composition of Collaborations — An Example. In: Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology), pp. 129–133. IEEE Computer Society Press, Los Alamitos (2006)
3. ISIS Project Website, `http://www.isisproject.org`
4. `http://www.sunspotworld.com`
5. `http://squawk.dev.java.net/`
6. Bræk, R., Haugen, Ø.: Engineering Real Time Systems: An Object-Oriented Methodology Using SDL. Prentice-Hall, Englewood Cliffs (1993)
7. Kraemer, F.A., Herrmann, P., Bræk, R.: Aligning UML 2.0 state machines and temporal logic for the efficient execution of services. In: Meersman, R., Tari, Z. (eds.) DOA 2006. LNCS, vol. 4276, pp. 1612–1632. Springer, Heidelberg (2006)
8. Bjerke, M.: Asynchronous Messaging between Embedded Java Devices. Project Thesis. Norwegian University of Science and Technology, Trondheim (December 2008)
9. Merha, B.T.: Code Generation for Executable State Machines on Embedded Java Devices. Project Thesis. Norwegian University of Science and Technology, Trondheim (December 2008)
10. Kraemer, F.A., Bræk, R., Herrmann, P.: Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, pp. 166–185. Springer, Heidelberg (2007)

11. Kraemer, F.A., Herrmann, P.: Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In: Ehring, K., Giese, H. (eds.) Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007). Electronic Communications of the EASST, vol. 7. EASST (2007)
12. Arseneau, E., Engelhardtsen, F.B.: Project playSIM: Experimenting with Java Card 3 System Programming. In: JavaOne (June 2009)
13. Garg, V.K.: Elements of Distributed Computing. John Wiley & Sons, Inc., New York (2002)
14. Chandra, T.D., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM 43(2), 225–267 (1996)
15. Tanenbaum, A.S.: Distributed Systems: Principles and Paradigms. Prentice-Hall, New Jersey (2002)
16. Kraemer, F.A., Slåtten, V., Herrmann, P.: Engineering Support for UML Activities by Automated Model-Checking — An Example. In: Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques, RISE (2007)
17. Kraemer, F.A., Bræk, R., Herrmann, P.: Compositional Service Engineering with Arctis. Telektronikk 105(1) (2009)
18. Kraemer, F.A.: Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart (2003)
19. `http://ksoap2.sourceforge.net/`
20. Bræk, R., Gorman, J., Haugen, Ø., Melby, G., Møller-Pedersen, B., Sanders, R.: Quality by Construction Exemplified by TIMe — The Integrated Methodology. Telektronikk 95(1), 73–82 (1997)
21. Olsen, A., Færgemand, O., Møller-Pedersen, B., Reed, R., Smith, J.R.W.: Systems Engineering Using SDL-92, Chapter 6 – Systems Engineering. Elsevier North-Holland, Inc., Amsterdam (1994)
22. Telelogic: Tau 4.4 User's Manual. Malmö (2002)
23. Kuhn, T., Gotzhein, R., Webel, C.: Model-Driven Development with SDL - Process, Tools, and Experiences. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 83–97. Springer, Heidelberg (2006)
24. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc., New York (1994)
25. D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML: the Catalysis Approach. Addison-Wesley, Reading (1999)
26. Reenskaug, T., Wold, P., Lehne, O.A.: Working with Objects, The OOram Software Engineering Method. Prentice-Hall, Englewood Cliffs (1995)
27. Jarvinen, H., Kurki-Suonio, R., Sakkinen, M., Systa, K.: Object-Oriented Specification of Reactive Systems. In: Proceedings of the 12th International Conference on Software Engineering, pp. 63–71. IEEE Computer Society Press, Los Alamitos (1990)
28. Fliege, I., Gotzhein, R.: Automated generation of micro protocol descriptions from SDL design specifications. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, pp. 150–165. Springer, Heidelberg (2007)
29. Herrmann, P., Krumm, H.: A Framework for Modeling Transfer Protocols. Computer Networks 34(2), 317–337 (2000)