Design and Compilation of an Object-Oriented Macroprogramming Language for Wireless Sensor Networks

Felix Jonathan Oppermann*, Kay Römer*, Luca Mottola[†‡], Gian Pietro Picco[§], Andrea Gaglione[¶]
*Graz University of Technology, Austria
Email: {oppermann,roemer}@tugraz.at
[†]Politecnico di Milano, Italy
[‡]SICS Swedish ICT
[§]University of Trento, Italy
[¶]Imperial College London, UK

***Abstract*—Wireless sensor network (WSN) programming is still largely performed by experts in a node-centric way using low-level languages such as C. Although numerous higher-level abstractions exist, each simplifying a specific aspect of distributed programming, real applications often require to combine multiple abstractions into a single program. Using current programming frameworks, this represents a difficult task. In previous work, we therefore defined a conceptual framework that facilitates abstraction composition by defining sound compositional rules among few fundamental abstraction categories. The framework is extensible: programmers can add new abstractions within the boundaries determined by the compositional rules.**

In this paper we describe the design of a language—called MPL—that instantiates this conceptual framework. To support the extensible nature of the framework, the language is object-oriented, which allows programmers to add new abstractions by inheriting from existing classes that implement predefined interfaces. We modeled the syntax after Java, to make it more palatable to inexperienced embedded programmers. Compared to Java, we modified the language to enable efficient execution on WSN devices. We designed and implemented a compiler that translates MPL language into executable C code, which spares the overhead of a virtual machine. By comparing MPL implementations against functionally-equivalent Contiki/C implementations of several benchmark applications, we determined that the performance overhead of MPL is limited, and yet the programming task is simplified.

***Index Terms*—Compilers, Java, object-oriented languages, wireless sensor networks.**

I. Introduction

During the last years, the use of wireless sensor networks (WSNs) significantly increased [14] while WSNs also started to make their way into commercial and industrial real-world applications. Nevertheless, a more widespread WSN adoption is still hampered by the unavailability of easy-to-use development tools. As of today, most WSN applications are still implemented in low-level C code and their design requires in-depth knowledge of the specifics of embedded systems and low-power wireless communication. Consequently, WSN programming is usually carried out by WSN experts. To gain more widespread use, WSN development needs to be more accessible to domain experts and programmers without a strong WSN background.

This requires a move from the still prevalent node-centric programming model towards a more holistic view of the network that hides low-level details. To this end, a growing number of macroprogramming abstractions have been designed that simplify programming of a specific distributed computing aspect (such as assigning roles to nodes [7] or defining a subset of nodes to communicate with [11]) by offering a domain-specific language. However, integrating multiple of these abstractions into a single program is still difficult. There also exist a number of macroprogramming languages that include a fixed set of abstractions, but new abstractions cannot be added easily.

In the `make`*Sense* [3] project we have therefore analyzed existing WSN abstractions, classified them based on few fundamental dimensions, and developed a conceptual framework that allows the composition of arbitrary abstractions according to predefined sound rules. The framework is also extensible in that abstractions that were not known by the time the framework was designed can be added later.

In this paper, we describe how this conceptual framework can be instantiated by means of a concrete language and we also describe a compiler to translate that language into efficient executable code. To sustain the extensibility of the conceptual framework, the language is object oriented and inspired by Java and thereby easy to use for programmers familiar with Java or C++. The language differs from Java to support efficient compilation to resource-constrained WSN devices and to support the extensibility of the above conceptual framework. The compiler offers a plug-in interface to support addition of new abstractions and automatically distributes application functionality among the gateway and the sensor nodes.

The increased abstraction level and the use of powerful programming abstractions enables a reduction of user-written code by more then 50% in comparison to an implementation

in low-level code for a set of typical applications.

The remainder of the paper is organized as follows. Section II briefly reviews the current state of the art. Section III summarizes the make*Sense* abstraction framework. In Section IV we derive a set of requirements for a language implementing this framework. Section V introduces design decisions and implementation details that enable us to implement a language that is capable of meeting these requirements. Section VI introduces the architecture of the underlying compiler framework and provides a closer look to the plug-in interfaces that enable the required extensibility. Finally, the performance and overhead of the approach for a typical set of applications is evaluated and discussed in comparison with more conservative C-based implementations in Section VII.

II. Related Work

A multitude of different systems aims to raise the abstraction level of WSN programming by providing high-level macroprogramming frameworks. These systems either represent the WSN as a distributed database [10], or provide more sophisticated frameworks on-top of C [9] or with custom high-level languages [4], [13]. Database-like interfaces are usually limited to data collection applications, while more complex frameworks usually require the use of an unfamiliar language. Systems of both categories are usually monolithic and do not provide a well-defined interface to integrate application-specific abstractions. In this regard, MPL extends the state of the art by providing an extensible macroprogramming framework that supports in-network control logic and is based on Java, a widespread programming language.

Nevertheless, our macroprogramming language also differs from the standard Java ME [15] framework by targeting even more resource constrained devices. In addition, Java ME does not provide WSN-specific extension points to integrate existing concepts that abstract from typical WSN challenges such as communication and distributed data processing. In comparison to standard Java, WSN-specific extensions significantly increase the utility of our macroprogramming language. Some low-resource Java virtual machines also exist that are targeted at low-power embedded and networked systems [1], [2], [17], but they still require a comparatively significant amount of resources. For example, the Squawk virtual machine uses 80 kB of program memory and consequently targets more powerful ARM-based embedded platforms [18]. Our approach differs in that it generates customized C code which is in turn compiled into optimized machine code for the intended target platform hence reducing the introduced overhead.

III. The make*Sense* framework

This section describes the conceptual WSN abstraction framework developed in the make*Sense* project [3]. Previous research has demonstrated that a large number of WSN programming abstractions exist that solve common challenges of WSN programming [12]. WSN programming abstractions tackle issues such as node addressing, definition of communication patterns, and distributed data processing. A typical example of such programming abstractions is Logical Neighborhoods [11], a system that allows to define groups of nodes based on their state and to communicate with them in a way similar to sending broadcast messages to physical neighbors. Programming abstractions, like Logical Neighborhoods, already simplified WSN programming, but they were difficult to combine with other abstractions in a single program.

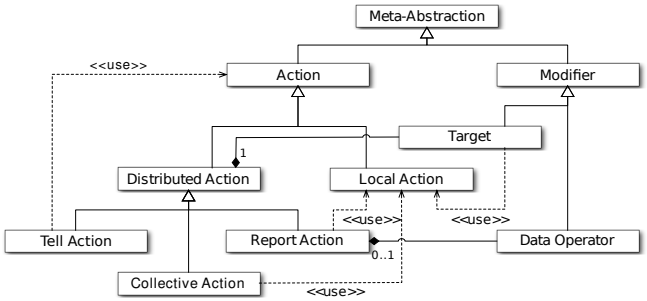

Fig. 1. The make*Sense* meta model for programming abstractions [3].

The make*Sense* project improved this situation by providing a unifying framework in which existing and future WSN programming abstractions can be easily integrated. To allow such extensibility, the make*Sense* framework employs the concept of meta-abstractions. WSN programming abstractions can be grouped into classes that aim to solve similar issues and that usually expose very similar interfaces. We call such a group of similar abstractions a "meta-abstraction", i.e., an abstraction of abstractions. Based on this insight, a hierarchy of typical WSN programming abstractions was developed in the make*Sense* project [3]. This hierarchy forms the basis of the make*Sense* framework as displayed in Fig. 1.

Two major types of *Meta-Abstraction* can be distinguished. *Actions* represent anything a node or a set of nodes can execute. This can be simple commands, like reading a sensor value, or more complex operations such as requesting aggregated values from a group of nodes. *Modifiers* allow a more precise specification of the behavior of Actions. For example, a Modifier could be used to specify which nodes should be part of the group that provides the aggregated value.

The Action class is further divided into two subclasses of actions. *Local Actions* are executed locally on a single node to implement basic operations, e.g., reading a sensor. In the make*Sense* framework, Local Actions also define the interface to the node hardware. Hardware operations, like reading sensor values or storing data on flash, are exposed to the user as Local Actions. In addition to predefined Local Actions, the user may also define custom Local Actions within the framework.

Distributed Actions can be used to request some action from multiple remote nodes. Distributed Actions typically define some form of communication between the nodes. The make*Sense* meta-abstraction hierarchy distinguishes between three basic communication patterns. The *Tell* abstraction is used for one-to-many communication. It allows to execute

other Actions on a set of remote nodes. Dually, the *Report* abstraction is used for many-to-one communication, e.g., to request sensor values from a set of nodes. On the remote node, the requested data is extracted by employing a Local Action that has been associated with the Report. Finally, *Collective Actions* provide a default interface for abstractions that implement peer-to-peer coordination patterns, such as global assertions [5] or role assignment [7].

Programmers can customize an action's behavior by using Modifiers, e.g., to select the set of nodes that participate in a Report. This separation of concerns enables a more flexible interaction among abstractions. The `makeSense` framework provides two subtypes of modifiers. *Target* modifiers are used to implement the aforementioned selection of nodes, to limit the scope of operation of any Distributed Action. *Data Operators* can be used with Report actions to define additional data processing to be applied to the collected data, such as aggregation.

The different meta-abstractions define extension points that can be instantiated by concrete implementations of these abstraction types. For example, the Target meta-abstraction could be instantiated by Logical Neighborhoods or another group-defining programming abstraction. Several instances of the same abstraction may exist at a time and an application may employ different abstractions instantiating the same meta-abstraction concurrently.

IV. Language Requirements

To implement actual applications, several abstractions need to be combined and augmented with application-specific functionality. This requires a programming language that allows to define the interaction between individual programming abstractions and to implement algorithms for custom data processing. A number of fundamental requirements for such a language can be defined:

(I) The language needs to be suitable to reflect the previously described meta-abstraction hierarchy supporting also later addition of instances of the abstraction classes.

(II) As some programming abstractions employ sophisticated custom languages for their configuration, the language requires a convenient mechanism to integrate such domain-specific languages.

(III) Individual nodes may need to handle several remote actions at once. Consequently, the language needs to support concurrent execution of tasks.

(IV) The language needs to be expressive enough to implement moderately complex algorithms, it needs to support at least basic mathematical and logical operators, conditionals, and basic looping constructs.

(V) The programming language should be familiar and easy to use for a large number of programmers. It should be especially appealing for typical domain experts.

(VI) Last but not least, the language needs to be adequate to generate efficient code suitable for considerably resource-constrained devices, such as wireless sensor nodes.

V. Detailed Language Design

Based on these requirements and the `makeSense` framework, we designed MPL, a high-level macroprogramming language for WSNs. As determined by Requirements IV and V, a goal for this programming language is to provide an expressive programming environment that is familiar to a large set of programmers. This made Java [8] a natural choice as basis for the design of MPL as it is well-established and widely used language with appropriate features. Building upon Java's object-oriented model also provides a stepping stone to implement the `makeSense` meta-abstraction hierarchy in accordance with Requirement I. Each meta-abstraction maps to an interface and abstractions can be added by implementing a meta-abstraction interface. The language is purely object-oriented with the exception of a limited set of primitive data types, including integers, chars, and Booleans.

To make the language suitable for resource-constrained devices as demanded by Requirement VI, some limitations compared to standard Java were necessary. The most significant difference is the absence of a virtual machine. Programs are instead translated into C code targeted at the Contiki platform that can be further processed by the established tool chain to generate deployable binary images.

Support for object-orientation in MPL had to be implemented in C in a standard-compliant way relying on structures and function pointers. The approach taken is conceptually similar to the approach taken by C++ [19]. Most notably, dynamic dispatch is implemented with the help of virtual method tables instead of relying on the more flexible but also more memory-demanding hash-table-based approach typically found in Java implementations.

In the following, we highlight other important design decisions that enabled us to meet the aforementioned requirements in an efficient manner.

A. Memory Model

Most object-oriented languages primarily employ dynamic memory allocation for objects and even though it is convenient for the programmer—especially if supported by garbage collection—it is ill suited for memory-constrained devices and reduces the efficiency of the program.

Therefore, to meet Requirement VI, MPL encourages the use of static memory allocation. In contrast to Java, it is not only possible to allocate new objects on the heap, but instead the language also provides additional operations to support different allocation schemes. Like in Java and in contrast to C++, objects are always accessed via references. To support alternative allocation strategies, we introduce two new allocation operators: `static` and `auto`, which return a reference to a static global object or an object allocated on the stack.

An object created with `static` is available for the complete run-time of the application. For example, the allocation of the rectangle `rec_global` in Listing 1 employs this feature. Static objects are represented by global variables in

```
class Rectangle {
  Point topLeft      = auto Point();
  Point bottomRight  = auto Point();
}

...

Rectangle rec_local   = auto Rectangle();
Rectangle rec_global  = static Rectangle();
```

Listing 1. Exemplary use of the extended memory model of MPL.

the generated C code. Their memory is statically allocated as part of the program image.

The operator `auto` allows to allocate objects on the stack. These objects are automatically deleted if the current block enclosed by `{` and `}` is left. If the newly allocated object has only been assigned to variables that are declared in the same block, then it behaves essentially like an automatic variable of a primitive type. Automatic objects are represented by local variables in the generated C code. The compiler also takes care of inserting appropriate statements in the C code to finalize objects before they are deleted.

The operator `auto` can also be used in a second role within the initializer expression of member variables. If used in an initializer, `auto` ties the lifetime of the newly created object to that of the host object. The class `Rectangle` in Listing 1 demonstrates this use of the `auto` operator. The memory required for the two `Point` instances is automatically allocated if a new instance of the `Rectangle` class is created. This type of automatic allocation is implemented by directly embedding the representation of the dependent object in the C representation of the host object. Consequently, the required storage space becomes part of the memory demand of the host.

To provide programmers with additional flexibility and to allow the creation of dynamic data structures, MPL also supports dynamic memory allocation and the standard `new` operator. Nevertheless, garbage collection is not supported, as this would add a significant run-time overhead. Dynamically allocated objects need to be destroyed explicitly by using a newly introduced `delete` operator. Dynamic memory allocation currently relies solely on the `malloc` implementation of the underlying target platform without additional optimization.

B. Multithreading

For the intended usage scenarios of MPL it is often necessary to execute several tasks in parallel. To support this in a user-friendly way, multithreading functionality is needed for the language as also indicated by Requirement III. The multithreading capabilities of MPL can be accessed via an interface that is closely modeled after the Java thread interface.

The current implementation is based on the Contiki multithreading library that provides a platform independent interface to switch the current stack. This interface is implemented for all major hardware platforms supported by Contiki. Based on this library, we implemented a custom thread scheduler running as a concurrent Contiki process that schedules runnable threads in a round robin fashion.

```
code neighborhoodDef = {:
  neighborhood HighTemperature() {
    System.getRole() == "sensor"
        and System.getType() == "temperature"
        and System.getTemperature() > x
  }
:};

Target highTemperature = auto LN(neighborhoodDef);
highTemperature.bindFloat("x", 30.0);

Report temperatureStream = auto Stream();
temperatureStream.setTarget(highTemperature);
temperatureStream.setAction(auto ReadTemperature());
temperatureStream.setDataOperator(auto MedianOperator());

temperatureStream.execute();
temperatureStream.waitResult();
Object result = temperatureStream.getResult();
```

Listing 2. Use of embedded code in MPL.

A major drawback of full-blown multithreading is the comparatively high memory overhead as each thread has its own stack that needs to be constantly kept in memory. To restrict the memory demand, the maximal number of concurrent threads is limited. If the maximum is reached, attempts to create further threads fail and an error is signaled to the user program. In the extreme case of an application that does not itself require multithreading, the whole application can be executed in a single thread. A separate thread running in parallel to the operating system is still required to support blocking operations without interfering with the operating system functions.

C. Embedding of Meta-abstractions

The core goal of the language design was the provision of an implementation of the `make`***Sense*** framework. As a consequence and in line with Requirements I and II, the language needs to be able to cleanly expose the abstraction-based extensibility features of the `make`***Sense*** framework.

To this end, the implementation of a programming abstraction within the language consists conceptually of three distinct components: (1) an MPL class, (2) a run-time module, and (3) (optionally) an MPL compiler plug-in. The MPL class serves as a means for MPL code to interface with the abstraction. Abstraction classes slightly differ from regular MPL classes in that their methods are typically not implemented by MPL code. Instead, each abstraction provides an additional run-time module that implements the required functionality in low-level C code. This increases efficiency and allows one to reuse existing implementations. In addition, the implementation of programming abstractions will typically be conducted by WSN experts that are already familiar with C programming on embedded devices. Finally, as indicated by Requirement II, programming abstractions may employ their own domain-specific language. To support a domain-specific language, the abstraction also needs to provide a compiler plug-in to translate the domain-specific code into code executable on the target platform. These plug-ins employ a compiler interface that is described in more detail in Section VI-B.

To support the use of domain-specific languages in a consistent way, MPL provides a dedicated syntax for *embedded code*. The mechanism for embedding code shares some similarities with prepared statements for embedding SQL code in programming languages. In contrast to prepared statements, embedded code fragments are not represented by strings, but by the special data type `code` to facilitate type checking and special handling by the compiler. Variables of this type can be instantiated only by a constant expression, consisting of a code fragment in an abstraction-specific language that is enclosed by the delimiters `{:` and `:}`. Also, unlike prepared statements, embedded code is not interpreted at run-time but compiled by abstraction-specific compiler plug-ins. As the embedded code is handled by plug-ins, it does not necessarily need to follow the syntax and semantics of MPL. To enable passing of values between embedded code and the MPL program, a binding mechanism is provided. Calls to a `bind` method of the affected abstraction allow one to bind variables in the embedded code to MPL variables.

To illustrate these aspects we employ Logical Neighborhoods [11] and a Stream abstraction as an example. A wrapper has been implemented in the `make`*Sense* project, that integrates an existing implementation of Logical Neighborhoods in the `make`*Sense* framework [3] by instantiating the Target meta-abstraction. Here, Logical Neighborhoods is used in combination with the Stream abstraction that implements the Report meta-abstraction. The Stream abstraction has been created from scratch within the `make`*Sense* project [3].

The Logical Neighborhoods abstraction employs a custom declarative language to define membership of nodes in a specific neighborhood, used as domain-specific language within MPL. This use of the functionality can be seen in lines 1–7 of Listing 2. In this example, a logical neighborhood `highTemperature` is defined that contains all nodes equipped with a temperature sensor and that read a high temperature value. The threshold value `x` is left as a parameter in the embedded code, and is later bound to the actual value in the main MPL program by means of an appropriate `bind` call in line 10.

The definition of the logical neighborhood relies on a set of node attributes ("role", "type", and "temperature") provided by the run-time environment. To allow simple access to such node attributes and operations in embedded code, these are exposed as static methods of a predefined `System` class. In this example, the `System` methods `getType` and `getTemperature` are used in the embedded code to determine the value of the corresponding attributes associated to the target node (i.e., the type of attached sensor and the sensor value). In line 9 of the example, a new instance of Logical Neighborhood is created, which is configured by the embedded code. This Logical Neighborhood instance is associated to a new Stream instance in line 11 to define the set of nodes from which the data should be streamed towards the sink. After also specifying a Local Action to read a sensor value and a Data Operator to aggregate the individual readings, the stream is started in line 17. Finally, the program waits for a result and stores the returned result in the `result` variable, as soon as available.

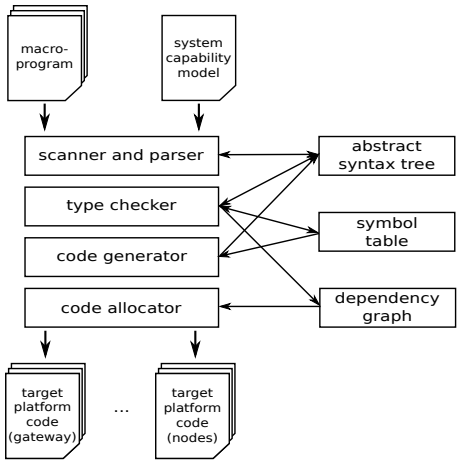

Fig. 2. Overview of the architecture of the MPL compiler. Arrows denote the data flow between these elements.

D. Interface to the Underlying Platform

MPL programs typically need to access functions of the underling hardware and software platforms. In MPL, such platform functions are also exposed through the previously introduced abstraction interface.

All communication-related tasks are handled by distributed actions. Direct access to functions to manually send or receive individual messages is intentionally not provided by the default implementation of MPL to shield the programmer from low-level details of communication. Nevertheless, such facilities could be provided by dedicated abstractions, if needed.

Node-local functions are exposed as Local Actions. This typically includes tasks like reading of sensor values and control of actuators, but the same facility could be also used to implement local data storage or data processing algorithms. We expect such platform-related functions to be implemented by WSN experts using C code with direct access to the underlying operating system. To support such an approach, methods in MPL can be declared as *native*, in which case the actual implementation of the method is provided by an external C implementation. Access to MPL language features is possible via a predefined C interface. Amongst other things, this interface provides the means to access, manipulate, and create MPL-defined objects within user-provided C code.

A number of Local Actions for commonly used functions are predefined by the language. As seen before, a subset of these functions is also available as static methods of an automatically generated `System` class to enable access from embedded code.

VI. MPL Compiler

Fig. 2 gives a high-level overview of the MPL compiler architecture[1]. Primary input to the MPL compiler is a macro-

[1]The compiler and supplementary software are available as part of the `make`*Sense* tutorial at http://project-makesense.eu/tutorial/makeSense-tutorial.zip

program written in MPL, possibly consisting of several source files. The macroprogram is supplemented by information about the system's capabilities, such as the hardware features and on-board sensors of deployed nodes. This information is used by the MPL compiler to aid optimization and the allocation of functionality to the different nodes. In addition to these inputs, the MPL compiler has access to a repository of components implementing the macroprogramming abstractions and run-time functionality. As output, the MPL compiler generates platform-specific source code for each node type of the target WSN, e. g., *gateway* and *regular nodes*, which is translated into a deployable binary image by the regular platform tool-chain. Our current prototype generates C-code for the Contiki operating system [6]. It is intended that later versions of the compiler will be extended with further code generators for different platforms.

A. Code Generation and Allocation

The compiler is implemented as a multi-pass compiler in Java. The compilation process consists of four distinct phases: parsing, type checking, code generation, and code allocation. All phases access the abstract syntax tree (AST) of the program and a shared symbol table. The implementations of parsing and type checking largely follow established approaches for compiler design. The code generation phase only differs from typical compilers in that it does not directly generate machine code. In the final code allocation phase, the compiler maps the compiled classes and interfaces to the available node types. A WSN may contain nodes with different capabilities that serve different purposes in the network. Not all of these nodes require the full functionality of the MPL program and part of the program is only ever executed on the nodes of a specific type. The code allocation phase allows one to remove unnecessary classes from the final code images and thus to reduce memory demands. In contrast to possible local code optimizations by the downstream C compiler, the allocation algorithm of the MPL compiler can take the entire control- and data-flow of the complete application, including remote invocations of abstractions, into account.

The current compiler prototype only distinguishes between a more powerful gateway node and the regular sensor nodes, but more complex allocation schemes are conceivable. The allocation procedure is based on the dependency graph generated by the type checker. For the gateway, the allocation process starts at the `main` method. This method is the central entry point of the MPL program and is always executed on the gateway machine. Starting from the `main` method, all classes used by this method are recursively collected. Only the compiled code of these classes is deployed on the gateway. For the nodes, the process starts at the set of actions that are defined in the MPL program. The compiler determines for each Action, if it is used by a Remote Action (e. g., Tell or Report). Only those Actions can be executed on a remote device and thus on one of the nodes in the WSN. For each of those Actions, all required classes are recursively collected. With the current prototype, this set of classes is deployed on all nodes. In the future, we will also take the capabilities of the nodes into account and build independent sets for each node type. In a second step, all Actions requiring capabilities that are unavailable at a specific node type are removed from the respective set.

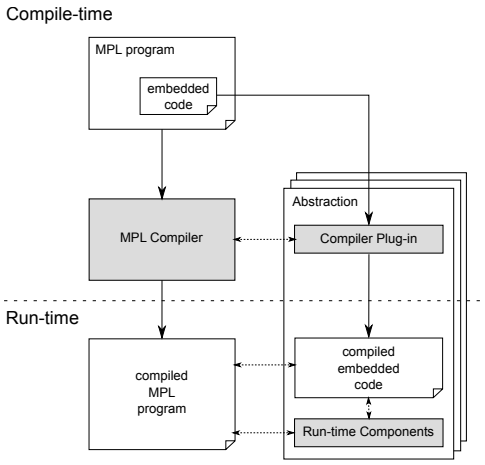

Fig. 3. Interaction of the MPL compiler and the compiler plug-ins. Continuous arrows represent data flow. Dashed arrows represent communication.

B. Plug-in Interfaces

Some programming abstractions employ embedded code to enable extensive configuration, as described in Section V-C. The abstraction-specific code cannot be handled by the MPL compiler itself, instead these abstractions need to provide compiler plug-ins to analyze and translate the abstraction-specific code. The MPL compiler plug-in interface allows these plug-ins to communicate with the MPL compiler at compile-time. Each compiler plug-in is essentially an independent little compiler with its own parser and code generator. Due to the fundamentally different nature of the object-oriented MPL code and the typically declarative embedded code, plug-ins cannot reuse the parsing and code generation functionality of the MPL compiler. Fig. 3 gives an overview of the interaction between the MPL compiler and the compiler plug-ins. Embedded code fragments are extracted by the MPL compiler and passed to the compiler plug-in provided by the abstraction that uses the code fragment. The compiler plug-in translates the domain-specific code into C code. The resulting C code is compiled with the native tool-chain and linked with the compiled binary image of the MPL program. The generated code can communicate with the abstraction-specific run-time component.

In some cases, it is necessary that the plug-ins generate different code for devices with different roles in the network. For example, the plug-in might differentiate between regular nodes and the network gateway. The gateway code might, for example, need to perform additional bookkeeping that is not required on the regular nodes. To support this, the plug-in may generate different C files for each of the available node types. The code is only linked with the correct binary image, in this case. Like the MPL compiler itself, plug-ins are implemented in Java.

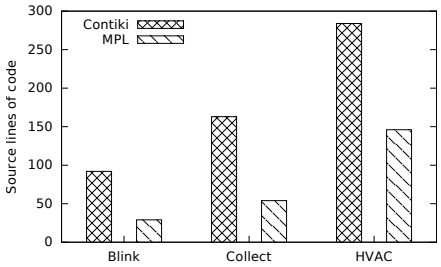

Fig. 4. User-written source lines of code per application.

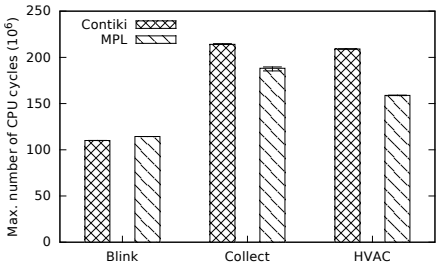

Fig. 5. Maximal number of CPU cycles spent on the nodes.

VII. Evaluation

To evaluate the performance of MPL we selected a small set of WSN applications and implemented each of them as a macroprogram and as functionally-equivalent Contiki/C programs. We then compared the performance of both implementations based on a set of typical software performance metrics for WSNs.

A. Settings and Metrics

We selected the following application scenarios for the evaluation:

1) *Blink to Radio.* This application scenario represents one-to-many communication. Such communication patterns are often used to distribute commands or configuration settings to a number of nodes. During run-time, the gateway process regularly sends a command to all sensor nodes requesting them to toggle their LED.
2) *Collect.* This application scenario consists of a simple data collection application. Data collection is a typical task for WSNs and as such is an important component of many real world applications. In the evaluated application, temperature and light sensor readings of all nodes are periodically sent to the gateway. At the gateway, the readings of all nodes are averaged and the result is reported to the user as a command line output.
3) *HVAC application.* The last application, originally developed in the `make`*Sense* project, implements a simple ventilation control system that regulates ventilation based on CO_2 readings. This scenario represents a simple real-world WSN application. Our setup consists of sensor and actuator nodes deployed in two rooms. The control process for each room is offloaded to one of the nodes located in the respective room.

All applications were implemented by an average programmer without a strong background in WSN programming. Before implementing the applications, he was provided with an introductory tutorial for the respective technology. During the experiments, the MPL and the Contiki/C programs was executed in identical simulated environments with five to six nodes and a gateway[2]. The WSN was simulated in real-time with Cooja, while the gateway code was executed in parallel in a 64-Bit Linux environment. The latter communicated with the simulated network via a serial socket and a simulated interface node [3]. Each application was executed until a scenario-specific termination condition was met. The same termination condition was used for the MPL and the Contiki/C implementation.

For each of the programs we investigated the following metrics:

1) The number of *source lines of code* is employed as an approximation of the programming effort for each application. We are aware that source lines of code are a very imprecise metric, especially if comparing different languages. Nevertheless, it can provide an initial idea of the relative complexities of the evaluated applications if consistent code formatting is used.
2) *Code size* is an important metric for WSN applications, as sensor nodes typically only possess limited program memory.
3) *Memory consumption* also needs to be kept low as random access memory is also a severely limited resource on sensor nodes. We need to distinguish the space required by statically allocated objects and the amount of heap space employed by dynamic memory allocation. Especially the use of dynamically allocated memory should be reduced, as dynamic memory allocation requires significant over-provisioning of memory resources. We measure heap allocation by linking the applications to a modified `malloc` implementation that tracks heap usage. Effects like fragmentation are not taken into account.
4) To compare the *computational overhead*, we count the CPU clock cycles spent on a typical execution of an application in a controlled environment.
5) The *communication overhead* is assessed by recording the number and size of radio messages sent during application execution. Ins WSN, energy consumption is often dominated by wireless communication, so that this metric also provides some insight on energy consumption.

B. Results

Fig. 4 demonstrates that the plain Contiki/C applications require the user to write more than twice as much code to ac-

[2]Please note, that even though the simulated scenarios employed only a small number of nodes, this does not invalidate the obtained results. The values for most of the employed metrics are not affected by the number of nodes.

[3]The node programs were compiled for the Contiki Tmote Sky target with an MSP-enabled version of the GNU C compiler (version 4.7.0 20120322, mspgcc patch set 20120911). The gateway code was compiled with the GNU C compiler (version 4.6.3).

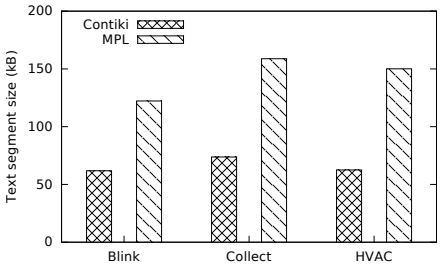

Fig. 6. Code size of the compiled gateway programs.

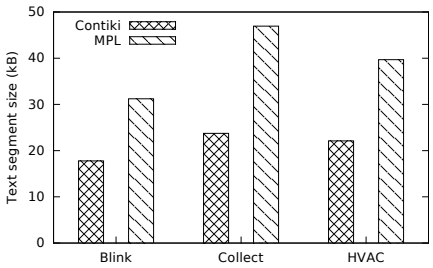

Fig. 7. Code size of the compiled node programs.

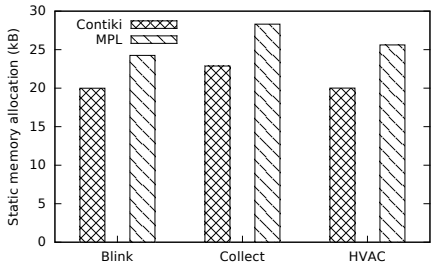

Fig. 8. Static memory allocation of the gateway programs.

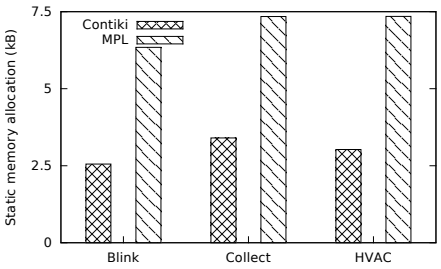

Fig. 9. Static memory allocation of the node programs.

complish the same task. In addition, further low-level technical details are exposed in the Contiki/C program. This indicates that the use of a high-level macroprogramming language like MPL can reduce the effort needed for the implementation of typical WSN applications. The programmer needs to write less code to achieve the same result.

Figs. 6 and 7 present the respective text size of the compiled MPL and Contiki/C implementations of the evaluated applications. It can be seen in Fig. 7 that the program image of the MPL programs is significantly larger for the evaluated applications. A cause for the increase in program size is the run-time environment required for some of the advanced features of MPL, like the support for multithreading. It should be noted that the overhead is largely constant (i.e., it does not grow with program size) and the relative overhead should be lower for larger, more complex applications. The program memory demand of the applications is still well within the limits of typical WSN platforms.

As presented in Figs. 8 and 9, the MPL-based code also makes use of more statically allocated memory. The overhead in static memory consumption is mainly caused by the additional data structures required for object-orientation support in MPL. As the relative overhead decreases with application size, we expect the relative overhead in the evaluated scenarios to represent a worst case scenario. More complex applications should exhibit a less significant relative overhead. Further optimization of the object representations can likely also significantly reduce the static memory demand of MPL-based programs.

While the Contiki/C-based programs do not make use of dynamic memory allocation, some features of the MPL programming model rely on dynamically allocated memory, e.g., to handle concurrent execution of Actions. This introduces a slight additional overhead, but the demand of dynamically allocated memory is comparatively little. None of the applications allocated more than 420 bytes at a time on any node in our test scenarios.

The computational overhead introduced by the use of MPL turns out to be very low in the experimental scenarios, as shown in Fig. 5. In the collect and the HVAC scenario, the MPL-based code actually employs less CPU cycles on the most active node of each network than the respective Contiki/C-based code. This demonstrates that features like virtual method dispatch do not introduce a significant overhead in terms of execution speed and energy consumption in typical applications.

As expected, the choice of a higher-level language does not significantly affect the total number of transmitted radio messages, as shown Fig. 10. Nevertheless, as shown in Fig. 11, the total amount of transmitted data is significantly higher for the MPL-based applications. This is mainly caused by the fact that the MPL code transmits complete objects that need to be serialized. An implementation providing a similar level of flexibility and features as the MPL code would be far more complex and consequently even more difficult to implement and maintain. At the same time its resource consumption would probably be much closer to the MPL-based implementation. Consequently, we conclude that the overhead for supporting a high-level macroprogramming language with object orientation is still reasonable for resource-constrained devices, like WSN nodes.

VIII. Conclusion

In this paper, we introduce the design and implementation of a Java-like macroprogramming language for the `make`*Sense* framework. A preliminary evaluation demonstrated that it is

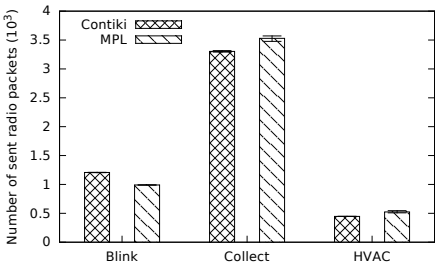

Fig. 10. Number of transmitted radio messages.

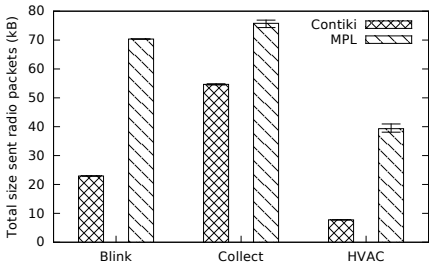

Fig. 11. Total size of transmitted radio messages.

possible to implement a high-level object-oriented macroprogramming language with a reasonable overhead.

Despite the positive results of the evaluation, current limitations open up an avenue for future work. The performance of the system can be further improved to make it even more suitable for the resource-constrained devices typically found in WSNs. Especially, memory consumption of the generated code still leaves significant room for improvements. Memory consumption could be, for example, improved by a more sophisticated strategy for code allocation. In the current implementation, code allocation operates at class level. Instead it would be possible to extend these decisions to individual methods and attributes. In addition, it would be useful to make the allocation algorithm work with a larger number of node types and to take the actual program behavior into account. To make the system more useful in practice, we also intend to improve debugging support. Finally, compatibility and dependencies among abstractions and between abstractions and the underlying protocols are not satisfactorily handled by the framework. The selection of suitable abstractions still requires manual intervention and some degree of expertise. Ideally, this selection would be largely automatic based on an abstract set of user-defined requirements. We currently explore possible solutions in the RELYonIT project [16].

ACKNOWLEDGMENT

Special thanks go to Niclas Finne for his support of the implementation of the MPL language run-time and compiler and to Bengt-Ove Holländer for his significant assistance with the implementation of the example programs used in the evaluation. The research leading to these results has received funding from the European Union 7^{th} Framework Programme (FP7-ICT-2009-5, FP7-ICT-2011-8) under grant agreements n° 258351 (`make`*Sense*) and n° 317826 (RELYonIT).

REFERENCES

[1] F. Aslam, L. Fennell, C. Schindelhauer, P. Thiemann, G. Ernst, E. Haussmann, S. Rührup, and Z. A. Uzmi, "Optimized Java binary and virtual machine for tiny motes," in *Proc. of the 6th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2010, pp. 15–30.

[2] N. Brouwers, P. Corke, and K. Langendoen, "Darjeeling, a Java compatible virtual machine for microcontrollers," in *Proc. of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, 2008, pp. 18–23.

[3] F. Casati, F. Daniel, G. Dantchev, J. Eriksson, N. Finne, S. Karnouskos, P. Montera, L. Mottola, F. Oppermann, G. Picco, A. Quartulli, K. Romer, P. Spiess, S. Tranquillini, and T. Voigt, "Towards business processes orchestrating the physical enterprise with wireless sensor networks," in *Proc. of the 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 1357–1360.

[4] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica, "The design and implementation of a declarative sensor network system," in *Proc. of the 5th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Nov. 2007, pp. 175–188.

[5] Ş. Gună, L. Mottola, and G. Picco, "DICE: Monitoring global invariants with wireless sensor networks," *ACM Transactions on Sensor Networks*, vol. 10, no. 4, pp. 54:1–54:34, Jun. 2014.

[6] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki – a lightweight and flexible operating system for tiny networked sensors," in *Proc. of the First IEEE Workshop on Embedded Networked Sensors (EmNets)*, Nov. 2004, pp. 455–462.

[7] C. Frank and K. Römer, "Algorithms for generic role assignment in wireless sensor networks," in *Proc. of the 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Nov. 2005, pp. 230–242.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java™ Language Specification*, 3rd ed. Addison-Wesley Professional, 2005.

[9] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," in *Proc. of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2007, pp. 200–210.

[10] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 122–173, Mar. 2005.

[11] L. Mottola and G. Picco, "Programming wireless sensor networks with logical neighborhoods," in *Proc. of the 1st International Conference on Integrated Internet Ad Hoc and Sensor Networks (InterSense)*, May 2006, p. 8.

[12] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Computing Surveys*, vol. 43, no. 3, pp. 19:1–19:51, Apr. 2011.

[13] R. Newton, G. Morrisett, and M. Welsh, "The Regiment macroprogramming system," in *Proc. of the 6th International ACM/IEEE Conference on Information Processing in Sensor Networks (IPSN)*, 2007, pp. 489–498.

[14] F. J. Oppermann, C. A. Boano, and K. Römer, "A decade of wireless sensing applications: Survey and taxonomy," in *The Art of Wireless Sensor Networks*, ser. Signals and Communication Technology, H. M. Ammari, Ed. Berlin, Germany: Springer, 2014, pp. 11–50.

[15] Oracle, "Java ME and Java Card technology." [Online]. Available: http://www.oracle.com/technetwork/java/javame/index.html

[16] RELYonIT Consortium, "RELYonIT: Research by experimentation for dependability on the internet of things," 2014. [Online]. Available: http://relyonit.eu

[17] N. Shaylor, D. Simon, and W. Bush, "A Java virtual machine architecture for very small devices," in *Proc. of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems (LCTES)*, 2003, pp. 34–41.

[18] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, "Java™ on the bare metal of wireless sensor devices: The Squawk Java virtual machine," in *Proc. of the 2nd International Conference on Virtual Execution Environments (VEE)*, 2006, pp. 78–88.

[19] B. Stroustrup, "Multiple inheritance for C++," *Computing Systems*, vol. 2, no. 4, pp. 367–395, 1989.