GENERIC IMPLEMENTATION FOR HETEROGENEOUS TINY ARTIFACTS

Tobias Baumgartner
Braunschweig University of Technology
Braunschweig, Germany
t.baumgartner@tu-bs.de

Alexander Kröller
Braunschweig University of Technology
Braunschweig, Germany
a.kroeller@tu-bs.de

1. Introduction

The Wiselib [BCF+10] is a generic algorithm library for heterogeneous tiny artefacts. The main objective is to provide algorithms that can run on different hardware and software platforms. Such an algorithm can be compiled for any supported hardware platform, and even for a simulator without changing any line of algorithm code. The library is written in C++, which allows the use of modern programming techniques—especially the use of templates, which are successfully used by well-known C++ libraries such as the STL or Boost. The basic idea in the Wiselib is to pass important functionalities, as template parameters to an algorithm: implementations of OS specific code and data structures. Hence, it is possible to compile an algorithm exactly for the current needs, without producing too much overhead—neither in code space nor in run-time.

The Wiselib supports several target platforms, including well-known low-end operation systems such as TinyOS or Contiki. This comes with a broad range of different kinds of artefacts—from tiny nodes with 8-bit microprocessors and tiny amounts of RAM, up to powerful 32-bit nodes being able to run desktop operating systems such as Linux. Even Smartphones—both iPhone and devices running Android—are supported, allowing for the interconnection of the mobile world with tiny artefacts. While it is easy to write code for a specific platform, it is a very challenging task to develop platform-independent code. Not only that different operating systems are in use, but different hardware platforms come also with different versions of compilers. This means that the developer is forced to spend a great deal of attention on low-level details, making the process of designing, testing and maintaining source code painfully complex and time consuming.

However, the Wiselib, an algorithm library for heterogeneous tiny artefacts, remedies this unfortunate situation. This is especially useful for prospective software designers coming from a theoretical side, where the development of distributed algorithms for many actual or hypothetical problems has grown into a research field of its own. There is a large variety of highly sophisticated algorithms for all kinds of tasks available, but with only few of them been tried in practice. A main reason for the lack of theoretical profound algorithms is the difficult implementation process, holding theoreticians off practical evaluation of their ideas. With the Wiselib, we bypass these difficulties, providing a powerful development framework towards a simplified implementation of algorithms on tiny artefacts.

We present the basic building-blocks of the Wiselib, and show that the flexibility of the design has barely any overhead—neither in code size nor in run-time; one can simply add new algorithms only by following the presented approach using the Wiselib interfaces. An algorithm, can then run on each supported device or simulation platform. We already achieved a state in which such an algorithm runs on heterogeneous sensor networks, and even more, networks in which some parts consist of virtual nodes running in a simulator [BBF+10].

2. Target Platforms

The Wiselib covers several devices of tiny artefacts, from platforms with low-end microcontrollers as the MSP430, e.g. on the Tmote Sky, to high-end devices such as Smartphones or the iMote2 being able to run an Embedded Linux. Table 1 shows an overview of the already supported platforms.

Hardware	Firmware/OS	CPU	Language	Dyn Mem	ROM	RAM	Bits
iSense	iSense-FW	Jennic	C++	Physical	128kB	96kB	32
SCW MSB-A2	FeuerWare	NXP LPC2387	C	None	512kB	98kB	32
SCW MSB430	SCW2, Contiki	MSP430	C	None	48kB	10kB	16
Tmote Sky/TelosB	Contiki,TinyOS	MSP430	C, nesC	Physical	48kB	10kB	16
G-Node	TinyOS	MSP430	nesC	Physical	116kB	8kB	16
MicaZ	Contiki,TinyOS	ATMega128L	C, nesC	Physical	128kB	4kB	8
iMote2	TinyOS	Intel XScale	nesC	Physical	32MB	32MB	32
iPhone, iPod	iOS	ARM	C++, Obj-C	Virtual	$\geq$8GB	$\geq$128MB	32
Smartphone	Android	ARM	C++	Virtual	$\geq$2GB	$\geq$64MB	32
Desktop PC	Shawn	various	C++	Virtual	unlimited	unlimited	32/64
Desktop PC	TOSSIM	(ATMega128L)	nesC	(Physical)	unlimited	unlimited	(8)

Table 1: Target platforms of the Wiselib. The columns refer to the type of microcontroller, the standard operating system, the programming language for it, what kind of dynamic memory is available, the amount of ROM and RAM, and the bit width.

The operating systems vary from system-specific implementations such as iSense and ScatterWeb to generic approaches such as Contiki, TinyOS, and Linux. With the different OSs, there also several programming languages in use. The iSense firmware has been developed in C++, whereas the ScatterWeb firmware and Contiki use plain C. There are also custom language in use: TinyOS uses the C extension *nesC* [GLvB^{+}03], applications for iPhones are written in Objective-C.

Support for dynamic memory allocation is likewise scattered. Only on a few platforms, `malloc()` and `free()` are available. If not, the size of all memory blocks must be known at compile time. iSense, on the other hand, comes with an own memory management implementation, proving the C++ operators `new` and `delete`. Similarly, there are additional modules for Contiki and TinyOS available. In Contiki, one can use *managed memory allocator* or *memb block memory allocator*, for TinyOS there is *TinyAlloc* available. However, such implementations do not support virtual memory as known from Desktop PCs. The memory management is done directly in the physical memory. Only with more powerful systems, such as Desktop PCs or Smartphones, virtual memory is available.

There are also significant differences in the amount of available memory, ranging from a few kilobytes to hundreds of megabytes and more on Smartphones. Finally, also the bit width varies with the amount of supported systems. The Atmel Atmegas are 8-bit microcontrollers, the MSP430 are 16-bit microcontrollers, whereas the rest are 32-bit microcontrollers.

3. The Wiselib

The core design pattern for the Wiselib are generic programming techniques that are implemented using C++ templates, which are successfully used by well-known C++ libraries such as the STL or Boost. Templates can be used to develop very efficient and flexible applications. The basic functionality of templates is to allow the use of generic code that is fully resolved by the compiler when specific types are given. Thereby, only the code that is actually needed is generated, and methods and parameters passed as template parameters can be accessed directly. The main idea of the Wiselib is that system functionality—both OS specific code and data structures adapting to the current platform—is passed via template parameters.

3.1 Architecture

The fundamental design principle of the Wiselib consists of concepts and models. There are basically three core components: OS facets, data structures, and algorithms. The idea is shown in Fig. 1.

OS facets are the connection to the supported operating systems. They represent basic system functionality,

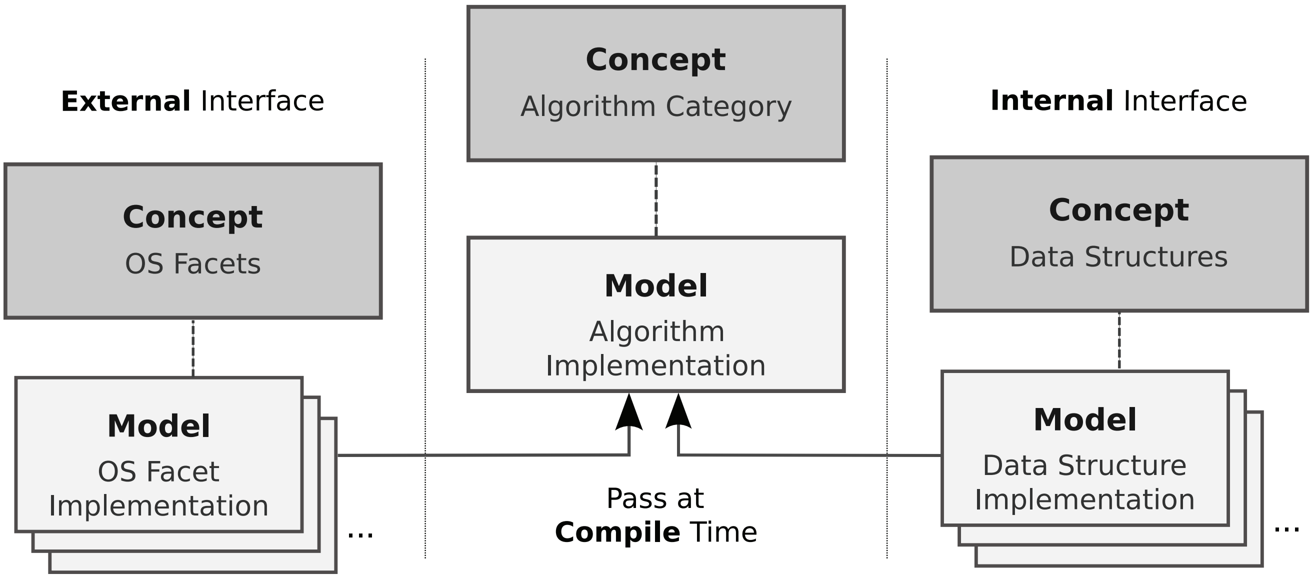

Figure 1: Wiselib Architecture.

such as sending messages over the radio interface, or registering callback events after a given time interval. They can be seen as a abstraction layer of the underlying OS, but held as simple as possible to ensure the efficiency of the Wiselib. Hence, there are mostly only type definitions and wrapper functions, no replication of OS functionality if not required.

Due to the range in supported hardware platforms, it is also possible within the Wiselib to use exactly the data structure that is suitable. There are static implementations of lists and maps, if no dynamic memory allocation is possible. But on high-end systems, efficient data structures, e.g. from the STL, can be used. Algorithm code is not affected by such a choice, since data structures are passed as template arguments.

Finally, the algorithms are the core of the Wiselib. There are several algorithms from several categories available, whereby a category groups algorithms by their fundamental functionality. For instance, there are categories for routing, clustering, and time synchronization algorithms. Algorithms do not contain any platform specific code, and can be used with each supported OS facet to let the algorithm run on any platform without changing any line of algorithm code.

A simplified example is shown in Fig. 2. There is one implementation of an algorithm that expects a radio as template parameter. If an implementation for Contiki is passed, the algorithm can run on Contiki-based systems. Alternatively, if an implementation for iSense is passed, the algorithm can run on these systems.

3.2 OS Facets

OS facets represent the connection to the underlying OS, and provide the basic functionality for an algorithm to work. There are several facets, for example a radio for sending and receiving messages, a timer for event registration, or a clock providing the current system time. The most important OS facets are implemented for all supported platforms, whereby different algorithms may require different facets. This comes with a big advantage: when integrating a new platform, this can be done by adding only a few facets with only a few hundred lines of code, and already having support for most of the algorithms in the Wiselib without too much effort.

OS calls in such facets can mostly be done by only doing a system call. This, can in turn be directly resolved by the compiler, resulting in no overhead, neither in code size nor in execution time. In some cases, for example in C-based operating systems, a translation between C++ member function calls and C function calls is required. This comes with a slight overhead, but as we see later in this Chapter, this overhead is very low.

Another advantage with the template-based approach of concepts and models is that one concept can be implemented by several models, each for special purposes. For example, we can provide one radio model for Contiki using the Rime stack, and another Contiki model using the 6lowPan implementation. When compiling an algorithm with either of these models, there is no overhead with this flexibility, since the compiler can resolve the

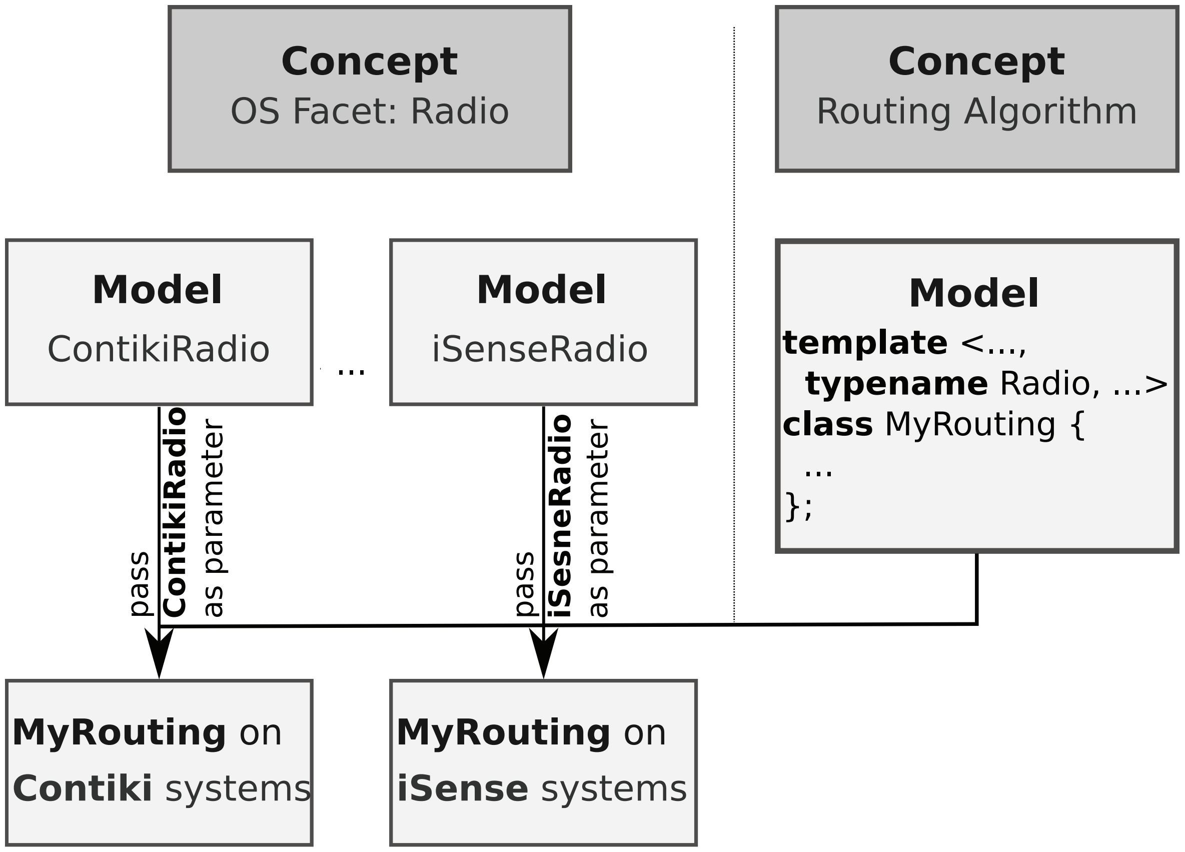

Figure 2: Passing a radio as template parameter.

corresponding code directly at compile time.

A reduced example for an iSense model of the OS facet "radio", written in C++, is as follows:

```
template<...> class iSenseRadioModel {
int send(id_t id, size_t len, data_t *data)
{
  os_.radio().send( id, len, data, 0, 0 );
  return SUCCESS;
}
```

In this example, the send-method of the radio is implemented, directly passing the call to the iSense firmware. The compiler can directly resolve this call, without generating any overhead.

3.3 Data Structures

There are target systems in the Wiselib, which do not provide dynamic memory allocation. Even worse, some compilers do even not support exceptions or RTTI. With these restrictions, it is not possible to use available data structures from an existing STL implementation. This affects both standard implementations as shipped with the GCC, as well as implementations designed for tiny devices such as the uSTL.

The Wiselib therefore provides an own STL implementation—the pSTL. Memory allocation is done statically, without any need for `malloc()` or `new()`. Furthermore, additional features such as exceptions or RTTI are not used. The pSTL is a subset of the STL, providing only the most important features. We have both concepts and models for the basic containers such as `list`, `set`, `map`, etc., as well as iterators and selected STL algorithms. Each Wiselib algorithm that awaits a model of the pSTL as template parameter can also be compiled with original STL data structures or compliant implementations, allowing for passing exactly the data structure that is convenient for the corresponding device. This results in a design that not only scales down to very limited devices, but also scales up to powerful nodes, utilizing all the available resources on them.

Another advantage with this approach is that the algorithm logic, which is invariant over different platforms, is decoupled from data storage, which heavily changes when an algorithm is ported to a platform of different characteristics.

3.4 Algorithms

The most important part of the Wiselib are the algorithms—enabling the execution of sophisticated protocols on different kinds of tiny artefacts, without changing any line of algorithm code. Again, Wiselib algorithms follow the design principle of concepts and models. There are several categories available, each with an own concept. For instance, we have a routing, a time synchronization, and a clustering category. Algorithm implementations (models) can belong to one or more categories—the latter common for cross-layer algorithms.

Such a model usually awaits both OS facets and data structures as template parameters. For a generalized usage of all algorithms within the Wiselib, there are common rules of how to initialize such elements. OS facets are initialized by the application that uses the algorithm, since there may be specific construction requirements on different platforms. Data structures, on the other hand, are initialized within the algorithm to encapsulate the knowledge of certain data values in the algorithm.

Having such well-defined algorithm interfaces comes with certain benefits. First, algorithms can be easily compared against each other, by simply altering a class name in the initialization code. This is useful for both testing purposes and evaluation issues. Second, all algorithms of a category are quite similar—at least regarding the basic type definitions and method declarations. Algorithm developers can thus start with a new model by copy-and-paste, in contrast to start from scratch. This lowers the difficulty of algorithm developments significantly, especially for theoreticians whose focus is on high-level algorithm evaluation instead of embedded development.

3.5 Advanced Design Techniques

The Wiselib design with concepts and models allows for an efficient realization of inheritance. Since we do not use any virtual inheritance, as known from ordinary classes in C++, we can easily derive one concept from another—without generating any overhead. The inheritance is only done in documentation, for instance, deriving an extended data radio concept, which provides additional data such as LQI or RSSI values for received messages, from the basic one. Any model can then implement this additional concept, and can be passed to algorithms expecting such an extended one. The compiler automatically generates the corresponding code, while not adding any unused functionality.

Another important design aspect is stackability, i.e., the possibility to build layered structures of multiple algorithms/facets. For instance, a routing algorithm provides the same methods and type definitions as a radio, a time synchronization algorithm implements the clock facet, and the position of a node can either be provided by a node's GPS, or a complex, distributed localization algorithm. Having such equivalence has a great advantage: Algorithms can be stacked together mostly free. An algorithm expecting a clock as template parameter, can be passed a complete time-synchronization algorithm, providing a global time basis for all nodes. Similarly, one can hide a routing algorithm behind the debug facet: All debug messages would not be written to the local UART, but automatically routed to sink collecting the debug output of the network. In both cases the algorithm would not be aware of such a behavior—it is completely managed in algorithm initialization.

Finally, message delivery in heterogeneous systems is a crucial problem. With the different kinds of hardware, there are basically three problems: different bit-widths, different byte-order, and differences with alignment. The Wiselib provides a sophisticated solution, again efficient due to template usage. The basic idea is to provide a serialization class with generic `read` and `write` methods, and then using template specialization to generate exactly the code matching for the current platform. Within these methods, only network byte order is used. The methods only support fixed-size data types, such as `uint16_t`, instead of data types where size changes with bit-width.

4. Experimental Results

We evaluated the efficiency of our generic algorithm library in two parts of the Wiselib: First, system call overhead and code size in OS facets; second, code size of selected algorithms for different platforms.

We compared Wiselib system calls with native OS calls on three platforms: iSense, Contiki on a TelosB, and ScatterWeb-FW on MSB nodes. TelosB and MSB are both equipped with a MSP430 processor. The results are shown in Table 2.

	iSense			Contiki			ScatterWeb		
	Native	**Wiselib**	**Cost**	**Native**	**Wiselib**	**Cost**	**Native**	**Wiselib**	**Cost**
Read ID	$2\mu s$	$2\mu s$	0%	$<1\mu s$	$<1\mu s$	0%	$<1\mu s$	$<1\mu s$	0%
Send Message	$282\mu s$	$282\mu s$	0%	$336\mu s$	$345\mu s$	3%	$898\mu s$	$921\mu s$	3%
Set Timer	$135\mu s$	$141\mu s$	4%	$77\mu s$	$100\mu s$	30%	$20\mu s$	$43\mu s$	115%

Table 2: Performance costs of Wiselib calls compared to native OS calls.

Both the read ID method on all evaluated platforms and the send message method on iSense systems are only one-liners, and can thus be directly inlined by the compiler. However, sending a message or setting a timer on Contiki and ScatterWeb requires the translation from C++ member function pointers to C function pointers, which in turn comes with a certain overhead. Similarly, setting a timer on iSense needs a translation between Wiselib and iSense specific OS callbacks. So there is a overhead in some Wiselib calls compared to native ones, but fortunately this is only in terms of microseconds. Moreover, any generic solution must come with a certain overhead, but the advantage of the Wiselib's solution with C++ and templates can remove such overhead where possible—see read ID method for an example.

Apart from call efficiency, code space is an important performance measure. As previously, we evaluated the iSense platform, Contiki, and ScatterWeb-FW. Evaluation was done with the radio facet and the timer—note again the platform differences: iSense runs on a 32-bit system, Contiki and ScatterWeb were compiled for 16-bit; with corresponding differences in size of machine language instructions. The results are shown in Table 3.

	iSense	**Contiki**	**ScatterWeb**
Radio	856+240	428+ 72	316+ 40
Timer	868+240	352+210	270+ 80

Table 3: Code-size overhead of OS facets. Shown is ROM (.text) and RAM (.bss + .data) in bytes.

Due to the fact that the concepts for radio and timer were held simple, no implementation required more than a few hundred lines of code. The slight structure led not only to enhanced maintenance issues, it also simplifies the integration of a completely new platform.

The resulting code size for all platforms is surprisingly small, given that, for example, C function pointer translation had to be done at certain points. In particular, the code size for the ScatterWeb platform is with around 600 bytes ROM, and 120 bytes RAM beneficial, given that there is not much more needed to run first routing or clustering algorithms. Furthermore, note that the ROM usage of the facets is constant, and does not grow with a higher amount of algorithms.

For the evaluation of algorithms, we took four implementations from the routing category: DSDV, DSR, a simple tree routing, and a flooding algorithm. We compiled each algorithm for several target platforms, from low-end TelosB running Contiki, up to the simulation environments Shawn and TOSSIM. Table 4 shows the resulting code sizes and initial RAM usage for the several platforms.

	16-bit OS		32-bit OS	Simulators	
Algorithm	**Contiki**	**ScatterWeb**	**iSense**	**Shawn**	**TOSSIM**
DSDV	1446+ 72	1466+ 72	4776+136	4351+ 4	19146+ 4
DSR	1964+338	1716+238	5396+356	6918+ 4	20845+ 4
Tree	920+ 16	724+ 14	4060+ 24	2974+ 4	9946+ 4
Flooding	1122+ 50	762+ 34	2864+ 68	2260+ 4	10192+ 4

Table 4: Evaluation of code size as ROM size (.text) and RAM size (.bss + .data) in bytes.

The algorithm implementations adapt perfectly well to the different target platforms. Again, we must differentiate between 16-bit and 32-bit systems, due to their different size of machine language instructions. The shown sizes are the pure demand of the algorithm, without considering the OS facets.