TinyXXL: Language and Runtime Support for Cross-Layer Interactions

Andreas Lachenmann, Pedro José Marrón, Daniel Minder, Matthias Gauger, Olga Saukh, Kurt Rothermel

Universität Stuttgart, IPVS
Universitätsstr. 38, 70569 Stuttgart, Germany
{lachenmann, marron, minder, gauger, saukh, rothermel}@ipvs.uni-stuttgart.de

Abstract— **In the area of wireless sensor networks, cross-layer interactions are often preferred to strictly layered architectures. However, architectural properties such as modularity and the reusability of components suffer from such optimizations. In this paper we present *TinyXXL* that provides programming abstractions for data exchange, a form of cross-layer interaction with a large potential for optimizations. Our approach decouples components providing and using data, and it allows for automatic optimizations of applications composed of reusable components. Its runtime representation is efficient regarding memory consumption and processing overhead.**

I. Introduction

Given the resource constraints typical of sensor networks and the properties of wireless communication that cannot be handled well by a strictly layered architecture, cross-layer interactions are often regarded as a necessity [1]. They are needed when developing complex applications for platforms with only a few kilobytes of RAM or running a sensor node for months with a single set of batteries. In component-based architectures cross-layer interactions can be performed even more easily [2], since there is no explicit notion of layers. If cross-layer interactions are used in an unbridled way, however, they can have negative effects on desirable properties of the software architecture [3]. They reduce modularity and limit the reusability of software components. We argue that cross-layer interactions have to be supported by programming language abstractions and system software in order to alleviate these negative side-effects. Therefore, we do not focus on specific instances of cross-layer interactions but provide a framework that reduces the effort when applying them. This approach allows for reusability of components and performs automatic optimizations at compile-time.

In previous work [4] we analyzed several nontrivial sensor network applications and identified different forms of cross-layer interactions: merging of components, replacement of system components, global variables, function calls, and data exchange. In this paper, however, we focus on programming and language support for cross-layer data exchange, i.e., data jointly used by components which are on possibly different logical layers. The term "data exchange" comprises two distinct sub-classes of cross-layer interactions: parametrization and data sharing. Parametrization is used to adapt the behavior of components with well-defined switches that change the functionality, execution path, etc. There is usually only one component that offers an interface for parametrization, and a series of components that use it to provide values. For example, in TinyOS the MAC layer component can be parametrized at a fine granularity (e.g., turn on acknowledgments, set the length of the preamble). For data sharing there is usually only one component that provides its data to a set of other components that might be interested in it. Here the shared data gives a view on the component's internal data (e.g., the neighbor list or the current routing parent for the routing component).

Data exchange is a technique that offers a large potential for optimizations. First, parametrization is essential to tailor system components to the specific requirements of an application. Secondly, because of data sharing there is no need to keep redundant data in stringently constrained RAM and to acquire it twice with possibly high energy costs (e.g., for sending messages).

In current programming languages such as nesC [5] data exchange is often implemented with function calls. As we show in Section III, such an approach creates considerable overhead for the developers, especially when the application evolves. In addition, it hinders component reuse because the components forming the application have to be optimized by hand in order to prevent separate components from providing the same data twice. Therefore, we have created *TinyXXL* ("**Ex**change of **Cross**-**L**ayer Data for **Tiny**OS") that provides programming language support for data exchange.[1] *TinyXXL* is composed of two parts: a compile-time and a runtime component. For compile-time support of data exchange, we extended the nesC programming language to include abstractions for data definition and exchange. At runtime this language extension is complemented by the *TinyStateRepository* that stores all cross-layer data and provides efficient access to it.

With *TinyXXL* and the *TinyStateRepository* we pursue the goal of creating language and system support for highly optimized applications while fostering component reuse and independent software development. First, we try to decrease the effort of application developers when exchanging data. Secondly, our approach automatically optimizes applications at compile-time by removing redundant data provision code and selecting a data provider that meets the non-functional

[1]First ideas on *TinyXXL* have been outlined in [4].

requirements of the data users best. Therefore, unharmonized reusable components can be combined to form an optimized application without any data redundancies. Thirdly, since most of the checks are performed at compile-time, there is only little runtime overhead associated with the *TinyStateRepository*. Finally, the *TinyStateRepository* incurs no RAM overhead compared to manually optimized applications and allocates often less RAM than unoptimized ones.

TinyXXL has been developed as a part of the *TinyCubus* project [6], whose goal is to ease the development of adaptive sensor network applications. If *TinyXXL* is used in the context of the *TinyCubus* framework, some optimizations usually performed at compile-time have to be done on the sensor nodes at runtime. This is necessary because at compile-time there is no global view of the application due to *TinyCubus*'s dynamic adaptation capabilities.

The rest of this paper is organized as follows. Section II gives a brief overview of the *TinyCubus* framework. Section III describes *TinyXXL*, our extension of the nesC programming language to support cross-layer data exchange. In Section IV we then present the *TinyStateRepository*, *TinyXXL*'s runtime representation. In Section V we evaluate both *TinyXXL* and the *TinyStateRepository* and describe their advantages. Section VI gives an overview of related work. Finally, Section VII concludes this paper and describes possibilities to further extend *TinyXXL*.

II. Overview of TinyCubus

In order to support the requirements of flexibility, adaptation and reconfiguration needed by typical sensor network applications, we are developing a generic reconfigurable system software for sensor networks called *TinyCubus*. *TinyXXL* and the *TinyStateRepository* have been created as part of this framework. In this section we give a brief overview of the architecture of *TinyCubus*. This description is based on previous work [7], [6] and is provided for convenience and completeness.

TinyCubus is implemented on top of TinyOS [8]. It consists of three parts: the Tiny Data Management Framework, which supports the adaptation of components, the Tiny Configuration Engine, which allows for the exchange and reconfiguration of components at runtime, and the Tiny Cross-Layer Framework, where *TinyXXL* and the *TinyStateRepository* are part of.

A. Tiny Data Management Framework

The Tiny Data Management Framework is a set of adaptation system components that also provides data management functionality. For each type of standard data management component such as replication, caching, hoarding, prefetching or aggregation, as well as each type of system component, such as time synchronization and broadcast algorithms, *TinyCubus* assumes that several implementations of each component type providing the same interface exist. The Tiny Data Management Framework is then responsible for the selection of the correct implementation based on the current information available in the system.

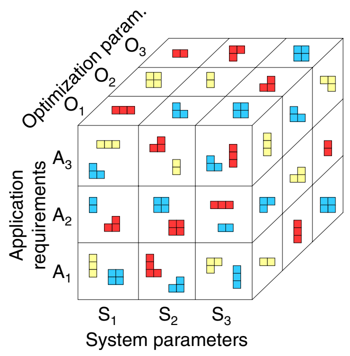


Fig. 1. Components in the Tiny Data Management Framework

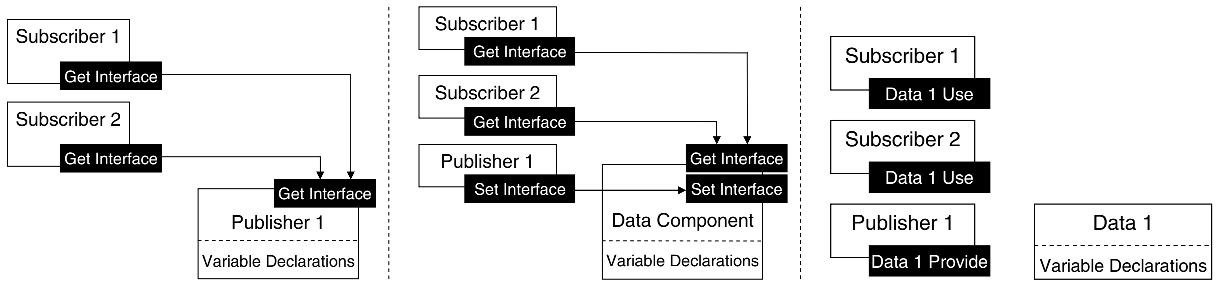
The cube of Fig. 1, called 'Cubus', combines optimization parameters $(O_1, O_2, \ldots)$, such as energy, communication latency and bandwidth; application requirements $(A_1, A_2, \ldots)$, such as reliability or consistency level; and system parameters $(S_1, S_2, \ldots)$, such as mobility or node density. For each component type, algorithms are statically classified in advance according to these three dimensions. For example, TinyAggregation [9] implements a tree-based routing algorithm used for aggregation in sensor networks that operates efficiently in static environments, but cannot be used effectively in highly mobile scenarios with strict reliability requirements. In such a setting, a flooding-based algorithm would probably do much better. Therefore, the component implementing the algorithm is tagged with the combination of parameters and requirements for which the algorithm is most efficient. The mapping of components to parameter values is performed off-line using experimental evaluations of each component in combination with the corresponding parameters. This way it is possible to know which components and/or groups of components perform best for a given parameter combination.

The Tiny Data Management Framework selects the best suited set of components based on current system parameters, application requirements, and optimization parameters. This adaptation has to be performed throughout the lifetime of the system and is a crucial part of the optimization process.

B. Tiny Configuration Engine

When new functionality such as a new processing or analysis function for sensed data is required by the application, it is necessary to install new components or swap functions. The Tiny Configuration Engine addresses this problem by distributing and installing code in the network. Its goal is to support the configuration of arbitrary components with the assistance of its two main parts: the *topology manager* and an in-place linking mechanism.

The *topology manager* is responsible for the self-configuration of the network and the assignment of specific functionality to each node. It also publishes topology information using the state repository that describes the neighborhood of sensor nodes, the status of communication links and the availability of components in other neighboring nodes. This and other cross-layer information contained in the *TinyState-*



(a) Data stored in publisher (b) Data stored separately (c) Data declared with *TinyXXL*

Fig. 2. Possibilities to declare shared data with nesC and *TinyXXL*

Repository can be used for the selection of more efficient routes for data and code dissemination, as shown in [7].

The linking mechanism allows for the reconfiguration of sensor nodes by providing the necessary bootstrapping code and the ability to load and install components on the fly. Using this approach the energy spent for code updates is largely reduced. In addition, it provides the flexibility needed for adaptation [10].

C. Tiny Cross-Layer Framework

The Tiny Cross-Layer Framework provides a generic interface to support the use of cross-layer interactions. It offers data exchange functionality with *TinyXXL* and the *TinyStateRepository*, which are described in the rest of this paper. The *TinyStateRepository* implements an efficient state repository that at runtime stores all data that is to be exchanged. The developer declares this data using *TinyXXL*, an extension of the nesC programming language, that facilitates cross-layer data exchange while largely preserving the modularity of components.

III. Data Exchange with TinyXXL

Current programming languages do not provide explicit and adequate support for data exchange. For example, nesC only allows the implementation of data exchange using function calls, which can lead to a significant development overhead and possibly unoptimized applications. First, the developer has to create an interface, implement this interface within a component, and "wire" all users of the piece of data to this component. Secondly, if two components provide the same piece of data, this data is stored and acquired twice, which might require energy-intensive operations such as sending messages. For example, if both MAC layer, routing, and application-level components maintain neighbor tables, they allocate limited memory space for redundant data. In addition, they waste processing time and energy, if they send beacon messages to update these tables.

Manually optimizing the components that form an application increases the development efforts significantly. With sensor network applications becoming more and more complex, inefficiencies due to duplicate data are hard to detect and even more so to fix. For example, TinyDB, an application providing a generic query interface, consists of almost 30,000 lines of code grouped in 176 components. Therefore, when developing such a complex application, it is difficult to detect components with duplicate data. It is even more difficult to optimize the application by removing data gathering code so that the same data is not stored and acquired twice. Often this code is not isolated in a special function but interwoven with other functionality needed for the component to work properly. In addition, as we describe in the following paragraphs, such modifications hinder reuse and independent development of components, two approaches often used to significantly decrease the development costs of software.

Basically, there are two common solutions to properly implement data exchange with nesC: The first one stores the data in the component sharing its data whereas the second solution uses a separate component to store the data that is wired to both its providers and subscribers.

Storing the data within the providing component (see Fig. 2(a)) is most often used in existing applications. However, this solution introduces tight coupling between components accessing and providing a given piece of data, which hinders maintainability. If a component providing some data is to be replaced, not only the wirings of its functional interface have to be adjusted but also those of the components that use its data. These wirings (the arrows in the figure) can be distributed across all application and system components. For example, an application component that does not send any radio messages may need to access the network neighborhood information of the routing component.

If data is stored in a separate component that has been exclusively created for data storage (such as in Fig. 2(b)), components accessing and providing some piece of data are decoupled; so the aforementioned maintainability problems do not appear with this solution. However, the compiler cannot guarantee automatically that there is a component in the system providing the data. In addition, to avoid duplicate data provision, publisher components have to check whether or not they have to acquire the data. This check is difficult to implement without runtime overhead, especially if non-functional requirements have to be considered. For example, such requirements could be a certain accuracy level or an update frequency needed by the data user.

We address these issues by extending nesC with *TinyXXL* in the following way:

```
xldata NeighborData(cost_type buildCost(<)) {
  NeighborNode neighborTbl[ROUTE_TABLE_SIZE];
  int8_t neighborCount;
}
```

Fig. 3. Declaration of shared data with *TinyXXL*

- *TinyXXL* decouples the components providing and using data (see Fig. 2(c)) by automatically creating wirings between them and by using a publish/subscribe scheme, which eases the process of data exchange.
- For shared data *TinyXXL* ensures that there is only a single component providing each data item. In contrast, with parametrization several components can provide values to modify the behavior of a specific one, such as the MAC layer component.
- *TinyXXL* adds capabilities for the specification of non-functional properties of data providers so that the system can select the one component that meets the requirements of data users best.
- *TinyXXL* provides efficient automatic notifications of subscribers after changes to the data.
- *TinyXXL* offers optimization capabilities that remove the data gathering code from all but one provider of a single kind of data. This way, no processing time and energy is spent for acquiring redundant data. Thus it is possible to develop optimized applications from reusable components without manual intervention. For example, both a generic MAC layer and a routing component can provide information about the quality of network links. With *TinyXXL* only one of them gathers this data; thus the size of allocated memory and – possibly – energy consumption are reduced.

A. TinyXXL Language Description

The changes to nesC needed to achieve the benefits mentioned above are relatively simple. Our additions and modifications include the ability to declare data definition files, specify data dependencies, specify **ifproviding** blocks for data publishers, and use virtual data items.

1) Data Definition: Within *TinyXXL* data is defined in a separate file similar to the way interfaces are specified in nesC. In such a file the developer groups all the data items that logically belong together. For example, an array with information about the neighboring nodes is declared in the same file as a counter for the number of elements in it (see Fig. 3).

The syntax of the definition of individual data items resembles the declaration of variables. Unlike interfaces, which can be implemented by several components, there is only a single instance of a data file. This way the data can be identified by its unique name. As the example above shows, it is possible to declare parameters for non-functional requirements (in this case “buildCost”), which are used by the system to select a publisher component that meets the requirements of the subscribers. The “less than” sign in the example expresses that the publisher with the smaller values for this parameter should be preferred if several ones fulfill these requirements. This structure provides hints to *TinyXXL* regarding possible optimization strategies. Depending on the kind of data, other requirements (e.g., concerning the update frequency of a data item or its accuracy) can be added.

```
module MultiHopRouter {
  provides {
    xldata NeighborData(COST_PERIODIC_MSG);
    ...
  }
  uses {
    xlparam RoutingParam;
    interface ReceiveMsg;
    ...
  }
}
implementation {
  event TOS_Msg* ReceiveMsg.receive(TOS_Msg* Msg) {
    ifproviding (NeighborData) {
      ...
      NeighborData.neighborTbl[iNbr].address
        = pRP->source;
      NeighborData.neighborTbl[iNbr].refresh
        = NBR_MOST_RECENT;
      ...
    }
  }
  event void RoutingParam.changed() {
    call Timer.stop();
    call Timer.start(TIMER_REPEAT,
      RoutingParam.updateInterval * 1024L);
  }
}
```

Fig. 4. Provision of data and use of parameters with *TinyXXL*

In the case of parametrization several components may set the values influencing a single parameter. Therefore, there is no need to select a publisher meeting non-functional requirements; such requirements have to be defined only for shared data.

2) Specification of Data Dependencies: Components exchanging data declare this property in their header similar to the interfaces provided. However, in contrast to interfaces there is no need to create wirings for data dependencies because they are automatically resolved by the *TinyXXL* compiler. If data definition files specify non-functional properties, components providing data have to give values for those here. Components subscribing to the data may specify an arbitrary condition that uses a data item’s non-functional requirements. This condition has to be met by the publisher component. For example, a data subscriber can specify that the cost for acquiring some data may not exceed a given limit and that the accuracy has to be better than some threshold.

Fig. 4 shows an example for a component publishing some data and subscribing to a parameter. From the developers’ point of view data is accessed like global variables. The difference is, however, that each variable name has to be preceded by the name of the data definition it is contained in (e.g., “NeighborData.neighborTbl”). Again, this is analogous to the use of interfaces in nesC. Another difference compared to global variables is that data accesses of publishers have to be included in an **ifproviding** block (see below).

If problems from concurrency can arise, developers have to enclose accesses to shared data in standard nesC **atomic** statements (not shown in the code example). Therefore, they can use the constructs that they are already familiar with from nesC.

Neither publishers nor subscribers can access the data via pointers to guarantee that only components declaring correctly their dependencies are able to access the data. Our experience after modifying and creating several nontrivial applications using *TinyXXL* shows that this limitation does not severely restrict the developer.

When a component subscribes to some data, the developers have to implement a special function named "changed" (see Fig. 4). This is a notification function called after data has been modified. If this functionality is not needed, the compiler removes this code. As we describe in Section III-C, data subscriptions themselves cannot be changed at runtime but are statically created at compile-time.

3) "ifproviding" Blocks for Data Publishers: For components publishing some data we added another language construct: the **ifproviding** blocks. All the code related to data provision has to be included in such a block (see Fig. 4). This is necessary for two reasons. First, if the component does not have to provide the data because another one supplies the same data, the code inside this block is removed by the compiler. So there are no unnecessary processing steps to provide the same data twice and no overhead at runtime to check if the component has to acquire the data. Secondly, at the end of such a block subscribers are automatically notified of the change. These notifications cannot be accidentally omitted by the developer and this solution offers higher efficiency than notifications after each variable assignment. There is no need for **ifproviding** blocks if a component just subscribes to some data; it may access the data anywhere in the code.

Obviously, code that is needed to fulfill a publisher's functional purpose cannot be encapsulated in an **ifproviding** block. Otherwise, the component could not work properly if another component publishes this data and the code within the **ifproviding** block is removed by the compiler. In this case the component also has to specify a dependency on the data as a subscriber so that it can access the data outside the **ifproviding** blocks. For example, a routing component that makes its internal neighborhood information available to other components also has to subscribe to this data if it uses it for routing.

4) Virtual Data Items: Besides data stored in RAM, we have added support for dynamically generated data with virtual data items. Virtual data items declare how the data can be computed dynamically from some other data already present. For subscribers, this is completely transparent; they cannot tell whether data is stored in RAM or generated on-the-fly.

Using virtual data items, operators known from databases can be implemented for cross-layer data. This functionality can include projections from several data definitions into a single

```
xlvirtual NeighborCountAggregator {
  provides xldata NeighborCount();
  uses xldata RoutingData();
}
implementation {
  xldata void NeighborCount.count(uint8_t* result) {
    uint8_t i;
    *result = 0;
    for (i=0; i<MAX_ELEMENT_COUNT; i++) {
      if (RoutingData.neighbors[i].flag != EMPTY)
        (*result)++;
    }
  }
}
```

Fig. 5. Sample virtual data item that aggregates the number of network neighbors

one and aggregation functions that perform some computation on the data. For example, Fig. 5 shows a virtual data item that aggregates the number of neighboring nodes by counting the elements in some other data structure. Similarly, if there is no data publisher for the kind of data needed by a subscriber in the system, virtual data can be used to convert some other data to the required format. For example, a component just interested in neighborhood information does not want to process complex routing information, although the requested data could be inferred from that. Therefore, a virtual data item can distill this information from the more complex internal data of the routing component. These conversion capabilities can also be used with evolving data definitions, as new versions of a component are developed. Here components still expecting the old data format can use virtual data instead of the actual representation in RAM. The decisions on whether to store some data in RAM or to provide it using virtual data is made by the *TinyXXL* compiler based on the publishers and subscribers available.

There are several advantages of virtual data items. First, it is possible to provide additional data besides just the internal representation of the publisher components. This way data that is not directly provided by the components in the system can still be used by subscribers. Secondly, using virtual data reduces the amount of data that is stored in limited RAM and does not impose the overhead of acquiring similar data twice. Thirdly, instead of requiring every publisher or subscriber to convert the data, the system provides the data already in an immediately usable format.

To implement a virtual data item, the dependencies on other data have to be specified just like with data accesses. Then for each variable declared in the represented data a function like the "NeighborCount.count" function in Fig. 5 has to be provided. Obviously, this function can incur some processing overhead. However, this processing overhead would also exist with conversions being implemented in pure nesC code and is often less than acquiring equivalent data twice.

B. Impact on the life cycle of applications

Using *TinyXXL* influences the whole life cycle of sensor network applications, including design, implementation, and operation. In the design phase the developer can select reusable components of which the application is composed without making sacrifices regarding cross-layer optimizations. In addition, despite the use of cross-layer interactions, the modularity of the application is preserved and components are decoupled. Thus, when the application evolves, components can be exchanged more easily.

In the implementation phase the developer does not have to manually optimize data exchange, e.g., by ensuring that no redundant data is gathered and stored. This is something that is already done by *TinyXXL*. Therefore, the implementation effort is largely reduced.

During the operation phase of a sensor network application *TinyXXL* helps in reducing resource consumption by automatically performing cross-layer data exchange. Although the developer does not have to deal with optimizations, the performance of an application built from reusable components is comparable to a manually optimized one, since no redundant data is acquired and stored.

C. TinyXXL Compiler

The *TinyXXL* compiler is a pre-compiler that outputs pure nesC code. It has been implemented in Java using JavaCC as a parser generator. From the data definitions and the data dependencies the *TinyXXL* compiler generates the files that implement the *TinyStateRepository* (see Section IV). It ensures that only components declaring their dependencies can access the data in the specified way. The compiler resolves non-functional requirements on publishers by adding preprocessor directives that select the data provider satisfying the requirements best.

Since the *TinyXXL* compiler resolves all data dependencies at compile-time, it is not possible that components subscribe to some data dynamically. We selected this approach for two reasons. First, our analysis of existing sensor network applications [4] did not show much need for dynamic subscriptions. Secondly, this approach does not incur any RAM overhead and notifications can be implemented more efficiently because there is no list of current subscribers to check.

The *TinyXXL* compiler translates all data accesses into function calls to the *TinyStateRepository*. This way it ensures that the data cannot be accessed via pointers. Like almost all nesC functions these function calls are later inlined by the nesC compiler so that the compiled code closely resembles direct variable accesses. Furthermore, each **ifproviding** block is translated into a regular if-statement that can be evaluated at compile-time. Thus none of the *TinyXXL* concepts imposes the runtime overhead of a function call.

IV. Runtime Support for Data Exchange

At runtime the *TinyStateRepository* provides support for cross-layer data exchange. It consists of a set of components generated by the *TinyXXL* compiler and stores the data and parameters specified using *TinyXXL*. For each data and parameter declaration the *TinyXXL* compiler creates a nesC component that declares data as variables within this component. This file also contains access functions to get and set the values of variables in a controlled way. So the code generated is similar to the example shown in Fig. 2(b). The subscribing components are automatically wired to the get interface and one of the publishers is selected to provide this data, which is wired to the set interface. The *TinyStateRepository* then automatically notifies all subscribers after the publisher has finished writing data.

In the *TinyStateRepository* the system keeps information about the name of the data item, its type of cross-layer interaction (parametrization or data sharing), a list of publishers of each data item, a list of subscribers, its data type, and the value of the shared data or parameter. Only the data values themselves are kept in RAM; all other information is translated at compile time and implicitly stored in the code image of the application.

The solution described so far requires the compiler to do most of the work like checking data dependencies. In addition, with its global view of the application the compiler is able to remove code that is not needed. However, such compile-time optimizations are not possible when using adaptation within the *TinyCubus* framework [6]. If such adaptation capabilities are to be supported, there is no longer a global view at compile-time, since binary components can be linked to the code image individually. However, the linker on the sensor node can be modified to do most of the optimizations during adaptation, which are performed by the compiler in the static case. For example, just like functional dependencies it can also check if all data dependencies are fulfilled and use the non-functional requirements to select a publisher that meets the requirements of the subscribing components best. By directly changing parts of the program code (i.e., the values of some constants) the linker can perform these changes without increasing RAM consumption and with only little runtime overhead for accesses to the *TinyStateRepository* (see Section V-D). The only difference is that, since *TinyCubus* does not compile any code on the sensor nodes, the code within the **ifproviding** blocks is not removed but simply not executed if it is not needed.

Storing data in the *TinyStateRepository* can even be of advantage for adaptation with *TinyCubus*. Since the sensor node has to be rebooted to install the new code, all the contents of RAM are lost during the adaptation process. However, storing the complete contents of RAM in non-volatile flash memory is not practical because the linker cannot necessarily relate an old memory location to a new one and thus cannot handle pointer variables, for example. Because of the development overhead it does not seem practical, either, to require each component to implement a serialization interface to store and load the component's data. However, with *TinyXXL* the data in the *TinyStateRepository* can be written to flash memory with little effort. As there are no pointers to data in the *TinyStateRepository*, no problems can arise from data changing its physical representation if a virtual data item is used after reboot instead of direct accesses to memory, for example. In addition, the compiler knows the data format and can generate appropriate serialization functions for almost all cases. In

TABLE I

COMPLEXITY OF SAMPLE APPLICATIONS

Application	# components	# LOC
Sense-R-Us	116	18,714
TinyDB	176	29,559
AcousticLocalization	69	11,359

TABLE II

NUMBER OF CHANGED LINES OF CODE (ADDED, REMOVED, AND MODIFIED) AND TOTAL NUMBERS IN THE MODIFIED COMPONENTS

Application	Changed # LOC			LOC mod. components
	Add.	Rem.	Mod.	
TinyDB	391	332	297	6,318
AcousticLocalization	40	24	8	1,468

summary, because of the serialization facilities offered by the *TinyStateRepository* the application can leverage data gathered before adaptation and thus can be fully functional again faster.

V. EVALUATION

In this section we evaluate both the benefits for developers using *TinyXXL* (i.e., the development costs and maintainability) and the run-time overhead on the sensor nodes regarding space requirements and processing.

A. Development Costs

To show the development costs of *TinyXXL* we have created a nontrivial application and modified parts of two other ones available from the TinyOS CVS repository at SourceForge.net[2] to use it. These applications contain between 11,000 and 30,000 lines of code (see Table I). Our newly created application is called Sense-R-Us; it uses a sensor network to determine the position of research assistants and to detect the location and duration of meetings. The modified applications are TinyDB and AcousticLocalization. TinyDB provides generic query processing capabilities whereas AcousticLocalization determines the geographic positions of nodes using the difference in the speed of radio waves and sound. In TinyDB our changes to use *TinyXXL* focus on the components related to communication. Nevertheless, the rest of the application also provides some opportunities to apply them.

Table II summarizes how many lines of code had to be added, removed, or modified to add data sharing and parametrization capabilities using *TinyXXL*. It also shows the total number of lines of code of all components that were modified. In TinyDB we have created 24 shared data variables and parameters, but in AcousticLocalization just 4 parameters, because in this application the components work mostly on internal data. Therefore, as Table II shows, in TinyDB by far more lines had to be changed. In general, the lines that were added are quite simple such as the variable declaration and the specification of data dependencies. Correspondingly, the lines of code that were removed were used to declare and share the variables with specialized interfaces. Modified lines of code were mostly changed so that accesses to shared data and parameters refer to the *TinyXXL* variables. One reason why the number of lines added is greater than those removed is that our modified versions publish more data than the original implementations because this data could also be useful for other components.

[2]`http://tinyos.cvs.sourceforge.net/tinyos/`

TABLE III

LINES OF CODE IN A MINIMAL APPLICATION AND NUMBER OF CHANGED LINES WHEN ADDING ANOTHER PUBLISHER

Version	Total # LOC	Changed # LOC		
		Add.	Rem.	Mod.
Pure nesC, data in publisher	46	1	9	1
Pure nesC, data separate	65	1	2	1
TinyXXL	39	1	0	1

If applications are developed from scratch with *TinyXXL*, the development costs are much smaller. In fact, there are fewer lines of code necessary than in pure nesC-based solutions. To show this, we have implemented a minimal application that shares a single variable between two components. We have created three versions of this application. Using just nesC, the first one stores the shared data in the publisher component (see Fig. 2(a)) whereas the second one stores it in a separate component (see Fig. 2(b)). Finally, the third version uses *TinyXXL* for data exchange (see Fig. 2(c)). All three variants offer the same functionality, use 20 bytes of RAM, and compile to 452 bytes of code.

Table III compares the code length of the different variants, which is one of the best predictors of understandability and maintainability [11]. The results in Table III show that the *TinyXXL* variant requires the smallest number of lines of code. In particular, it needs 40% fewer lines than the nesC approach that keeps the data in a separate component and still 15% fewer lines than the nesC variant storing the variable within the publisher component. Although these numbers depend on the number of shared data variables, the number of accesses to them, as well as the number of publishers and subscribers, this example gives some idea about what we expect to find in more complex applications.

Sense-R-Us shows that complex applications can be developed with *TinyXXL*. This application makes extensive use of data sharing. For example, during the first 60 seconds after power-on – when information about neighboring nodes is gathered – data from the *TinyStateRepository* is accessed almost 3,300 times. In retrospect *TinyXXL* has made developing this application much easier.

B. Maintainability

To show the benefits to maintainability, we modified the different versions of the minimal application introduced in Section V-A. We added another component that provides the same data as the publisher already present. We then tried to optimize the application so that only one of the publishers writes the shared data and that no redundant variables are stored in RAM. As shown in Table III, in all three versions of the application one line had to be added and one had to be

TABLE IV
CODE SIZE OF THE EXAMPLE APPLICATIONS (IN BYTES)

Application	**Original**	***TinyXXL***
TinyDB	62,144	61,894
AcousticLocalization	24,272	23,996

TABLE V
RUNTIME OVERHEAD OF *TinyXXL/TinyStateRepository*

	CPU Cycles		**Time** (μs)	
	Original	***TinyXXL***	**Original**	***TinyXXL***
Read access	6.49	8.05	0.88	1.09
Write access	10.19	10.34	1.38	1.40

modified. Besides that, in the two pure nesC versions code had to be removed manually because it would have been redundant. In the variant with the data stored directly in the publisher these were nine lines of code because the declaration of the redundant variable, all accesses, and the code providing the shared data to other components had to be removed. In the version that keeps the shared variable in a separate component only the accesses and the unused interface had to be deleted (two lines of code). In the *TinyXXL* version, however, nothing had to be changed.

C. Space Requirements

Table IV shows the size of the code in program memory for both the original applications and the ones built with *TinyXXL*. The numbers are almost identical because most function calls to the *TinyStateRepository* are inlined by the compiler and, therefore, the code is mostly equivalent. Due to slight differences in the implementation and the optimizations performed by the compiler there are some variations (about 1%). However, they are too small to derive a general trend.

TinyXXL does not increase the size of allocated memory in RAM. The data in the *TinyStateRepository* is the same as in pure nesC-based approaches, since the *TinyStateRepository* does not store any meta-data in limited RAM. If the application makes use of *TinyCubus*'s adaptation capabilities, there is no RAM overhead for the *TinyStateRepository*, either, since all information about data providers and subscribers is written to the code in program memory.

It should be noted that both TinyDB and AcousticLocalization have already been optimized by hand. Therefore, this setting is the worst case where *TinyXXL* is not able to add benefit via its optimizations. With less optimized applications than our sample applications (e.g., those built from reusable components), we expect both code size and RAM consumption to decrease since the *TinyXXL* compiler includes only one instance of the data in memory and removes redundant data gathering code.

D. Runtime Overhead

To find out the runtime overhead of *TinyXXL* compared to a pure nesC approach, we measured the number of processor cycles needed for data exchange. For this purpose we used ATEMU [12], an emulator for Mica2-based sensor networks. We instrumented both the original and our modified versions of TinyDB and ran both versions of the application for 60 simulated seconds. During this time the nodes periodically exchanged messages and updated their routing tables several times.

The numbers in Table V show that the overhead of *TinyXXL* for read accesses is about 24%. Although this overhead seems to be significant, it is not noticeable when running applications in practice; in absolute numbers it is just $0.21\mu s$. Furthermore, the speed of the processor and the clock cycles available are usually not regarded as a limiting factor in existing sensor network applications. In fact, compared to radio bandwidth the CPU is fast enough to process incoming messages without a need for receive queues [2].

We attribute the overhead of read accesses mainly to fewer optimizations performed by the nesC compiler. Although most function calls can be inlined, there are some situations where in the original version of TinyDB a variable is already stored in one of the processor's registers and does not have to be loaded again. The compiler does not always perform this optimization with data from the *TinyStateRepository* so that the average cycle count is slightly increased.

For write accesses the numbers of both versions of TinyDB are almost identical. In general, the compiler performs the same optimizations in all cases since it always has to write the value to the variable. Note that for this comparison the notification functions were not included, since they provide some additional functionality. In any case, the overhead of these functions is not associated with every write access but with **ifproviding** blocks because the subscribers are only notified once for each block. In our modified version of TinyDB, for example, about 2.5 write statements are enclosed in such a block on average.

If support for dynamic adaptation within *TinyCubus* is needed, the optimizations usually performed by the *TinyXXL* compiler are done by the *TinyCubus* linker. Since, compared to the number of data accesses, applications are only adapted infrequently and since the linking process already takes a few seconds [10], applying these optimizations is not performance-critical. However, with this approach there is also some extra overhead at runtime: for each **ifproviding** block an additional check is required, because the code of such a block cannot be removed by the compiler in this case. In TinyDB this overhead is just 3.94 cycles ($0.53\mu s$) on average. Because typically some computation or even the transmission of radio messages is included in such a block, the benefits of not executing this code – if it is not needed – clearly outweigh this overhead.

Even small processing overheads at runtime can sum up in the course of time and influence the lifetime of the sensor network. However, only if an application has been optimized, it provides better overall performance. Such manual optimizations are less and less feasible as applications become more complex and are increasingly developed by experts in the application domain rather than experts in sensor networks. Compared to unoptimized applications that gather the same kind of data twice, the small runtime overhead of *TinyXXL* and the *TinyStateRepository* can be neglected. For example, if *TinyXXL* avoids sending unnecessary radio packets, this alone will outweigh the energy consumed by the CPU for the *TinyStateRepository*'s small runtime overhead.

E. Advantages

The use of *TinyXXL* exhibits several advantages compared to pure nesC solutions.

- *TinyXXL* ensures the *modularity* of applications by having components explicitly declare their dependencies on shared data and parameters. Therefore, it is not necessary to directly wire components accessing some data to those that provide it.
- The components exchanging data are *decoupled* from each other and, when the application evolves, can be replaced independently.
- The *TinyXXL* compiler automatically *reduces resource consumption* by removing code that acquires redundant data and by selecting a single publisher component that fulfills the requirements of the subscribers with the least costs.
- Since redundant data is automatically removed from applications, components, which – in addition to their actual functional purpose – provide some of their data to other components, can be developed without wasting memory and energy. These components can then be *reused* in new applications, regardless of other components that possibly provide the same types of data.
- *TinyXXL*'s tight integration in a programming language allows us to perform *checks at compile time* without any runtime overhead and with only little overhead if support for dynamic adaptation is needed. For example, the *TinyXXL* compiler ensures important properties such as type safety as well as access control so that only components declaring their dependencies correctly on some data or parameter can access it.
- The integration in the programming language allows for an *efficient publish/subscribe mechanism without any RAM overhead* and with *automatic notifications of changes.*

VI. Related Work

Blackboard architectures [13] have been widely used in artificial intelligence systems. In this kind of architecture a blackboard is used to structure and store knowledge as a global database. Logically independent modules modify the data in order to incrementally solve a problem; they use the blackboard as the only form of interaction. Thus, similar to *TinyXXL* the modules are decoupled. However, there is no clear specification of data dependencies and all modules may modify the data. Furthermore, with *TinyXXL* there is still the possibility to employ the standard nesC forms of interaction.

In the realm of mobile ad-hoc networks (MANETs) the MobileMan project [14] creates a cross-layer architecture for the protocol stack. Although the concepts used for data sharing are similar to *TinyXXL*, MobileMan does not pursue the goal of easing the usage of cross-layer interactions and, therefore, is not part of a programming language. Furthermore, it assumes hardware platforms typical of MANETs, which are, in the general case, not so resource-constrained.

Part of the link abstraction SP for sensor networks [15] is a neighbor table data structure that can be accessed by protocols of several layers. However, with SP data sharing is limited to some *a priori* selected data items; unlike *TinyXXL* it does not allow for the definition of arbitrary shared data.

Köpke et al. [16] have created a publish/subscribe-based system for sensor nodes. However, their work does not provide many of the features of *TinyXXL* because it is not integrated into a programming language. For example, it does not guarantee type safety when accessing data and does not deal with multiple publishers providing possibly inconsistent, duplicate data. Unlike our approach this system needs some meta-data to be stored in limited RAM.

TinyGUYS [17] is a global data storage that deals with concurrency problems; it guards accesses to the variables by writing all changes in a buffer and having the scheduler copy this data to the real variables later. Such synchronization of accesses is not considered by our approach. We rather rely on the developer to add the standard nesC **atomic** statements when synchronization problems can occur. Obviously, TinyGUYS adds some memory overhead for the buffers which can be a problem with resource-constrained sensor nodes.

SNACK [18] is a configuration language, component library, and compiler based on the nesC language. Similar to our approach it tries to ease the development of efficient sensor network applications. However, it deals with the creation of service libraries that can be combined to form an application rather than focusing on the problems of cross-layer data exchange. Likewise, Hood [19] is a programming abstraction for nesC-based sensor networks that provides support for data exchange between neighboring nodes rather than components on a single node. Furthermore, Hood hides the cost of communication from the programmer which may lead to some additional message overhead. *TinyXXL*, in contrast, strives at optimizing applications and avoiding redundant messages for data acquisition.

VII. Conclusions and Future Work

In this paper we have described and evaluated *TinyXXL*, our extension to the nesC programming language for cross-layer data exchange. *TinyXXL* strives to ease the development of cross-layer optimized applications built from reusable components. It allows for the declaration of components' dependencies on data, decouples components providing and using some data, and automatically optimizes applications by avoiding redundant data provision. With *TinyXXL* data dependencies become first-class citizens similar to functional interfaces. *TinyXXL* is complemented by the *TinyStateRepository*, an efficient state repository generated by the *TinyXXL* compiler that stores all cross-layer data at runtime.

As shown in the evaluation, complex applications can be developed using *TinyXXL*. For example, we have modified TinyDB and AcousticLocalization, two nontrivial existing applications, to make use of *TinyXXL* and developed Sense-R-Us from scratch with our programming language abstractions. With our approach fewer lines of code are needed for data sharing and parametrization while profiting of additional benefits such as the selection of the "best" component to publish the data. We have also shown that the *TinyStateRepository* is an efficient implementation of a state repository that imposes no RAM overhead and only little runtime overhead.

Based on our experience with Sense-R-Us, we expect *TinyXXL* to encourage the use of cross-layer interactions in new applications. For newly developed components we anticipate that the number of data items provided grows even more in the future when all the benefits of *TinyXXL* are fully available within *TinyCubus*. For example, if data in the *TinyStateRepository* is automatically serialized during adaptation, this alone offers enough incentives to publish the internal data of components. As the number of data providers grows, we expect the number of components using this data to increase as well. We are convinced that *TinyXXL* will lead to a large number of reusable components that use cross-layer interactions for optimizations. This will allow for more reuse in sensor network application and decrease development overhead while – concerning RAM and energy optimizations – achieving similar results as manually optimized applications.

Regarding future work, we are still in the process of fully integrating *TinyXXL* in *TinyCubus*. In addition, we are exploring possibilities to combine *TinyXXL* with approaches such as Hood. So cross-layer data would not only be exchanged among components on a single node but also among neighboring nodes. Furthermore, we plan to make *TinyXXL* available to the public.

References

[1] A. J. Goldsmith and S. B. Wicker, "Design challenges for energy-constrained ad hoc wireless networks," *IEEE Wireless Communications*, vol. 9, no. 4, pp. 8–27, 2002.

[2] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, "The emergence of networking abstractions and techniques in TinyOS," in *Proc. of the 1st Symposium on Network Systems Design and Implementation*, 2004.

[3] V. Kawadia and P. R. Kumar, "A cautionary perspective on cross layer design," *IEEE Wireless Communications*, vol. 12, no. 1, pp. 3–11, 2005.

[4] A. Lachenmann, P. J. Marrón, D. Minder, and K. Rothermel, "An analysis of cross-layer interactions in sensor network applications," in *Proc. of the Second International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, 2005, pp. 121–126.

[5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proc. of the Conference on Programming Language Design and Implementation*, 2003, pp. 1–11.

[6] P. J. Marrón, D. Minder, A. Lachenmann, and K. Rothermel, "TinyCubus: An adaptive cross-layer framework for sensor networks," *it - Information Technology*, vol. 47, no. 2, pp. 87–97, 2005.

[7] P. J. Marrón, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel, "TinyCubus: A flexible and adaptive framework for sensor networks," in *Proc. of the Second European Workshop on Wireless Sensor Networks*, 2005, pp. 278–289.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 93–104.

[9] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: A tiny aggregation service for ad-hoc sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 131–146, Dec 2002.

[10] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "FlexCup: A flexible and efficient code update mechanism for sensor networks," in *Proc. of the Third European Workshop on Wireless Sensor Networks*, 2006, pp. 212–227.

[11] I. Sommerville, *Software Engineering*, 6th ed. Addison-Wesley, 2001.

[12] J. Polley, D. Blazakis, J. McGee, D. Rusk, J. S. Baras, and M. Karir, "ATEMU: A fine-grained sensor network simulator," in *Proc. of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, 2004, pp. 145 – 152.

[13] P. Nii, "The blackboard model of problem solving," *AI Magazine*, vol. 7, no. 2, pp. 38–53, 1986.

[14] M. Conti, G. Maselli, G. Turi, and S. Giodano, "Cross-layering in mobile ad hoc network design," *IEEE Computer*, vol. 37, no. 2, pp. 48–51, 2004.

[15] J. Polastre, J. Hui, P. Levis, J. Yhao, D. Culler, S. Shenker, and I. Stoica, "A unifying link abstraction for wireless sensor networks," in *Proc. of the 3rd International Conference on Embedded Networked Sensor Systems*, 2005, pp. 76–89.

[16] A. Köpke, V. Handziski, J.-H. Hauer, and H. Karl, "Structuring the information flow in component-based protocol implementations for wireless sensor nodes," in *Proc. of Work-in-Progress Session of the 1st European Workshop on Wireless Sensor Networks (EWSN)*, ser. Technical Report TKN-04-001 of Technical University Berlin, Telecommunication Networks Group, 2004, pp. 41–45.

[17] E. Cheong, J. Liebman, J. Liu, and F. Zhao, "TinyGALS: A programming model for event-driven embedded systems," in *Proc. of the 2003 ACM Symposium on Applied Computing*, 2003, pp. 698–704.

[18] B. Greenstein, E. Kohler, and D. Estrin, "A sensor network application construction kit (SNACK)," in *Proc. of the 2nd International Conference on Embedded Networked Sensor Systems*, 2004, pp. 69–80.

[19] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," in *Proc. of the 2nd International Conference on Mobile Systems, Applications, and Services*, 2004, pp. 99–110.