A Concurrency Abstraction for Reliable Sensor Network Applications

János Sallai[1], Miklós Maróti[2], and Ákos Lédeczi[1]

[1] Institute for Software Integrated Systems, Vanderbilt University, 2015 Terrace Place, Nashville, TN 37203, USA
{sallai,akos}@isis.vanderbilt.edu
[2] Bolyai Institute, University of Szeged, Szeged, Hungary
mmaroti@math.u-szeged.hu

Abstract. The prevailing paradigm in the regime of resource-constrained embedded devices is event-driven programming. It offers a lightweight yet powerful concurrency model without multiple stacks resulting in reduced memory usage compared to multi-threading. However, event-driven programs need to be implemented as explicit state machines, often with no or limited support from the development tools, resulting in ad-hoc and unstructured code that is error-prone and hard to debug. This paper presents TinyVT, an extension of the nesC language that provides a virtual threading abstraction on top of the event-driven execution model of TinyOS with minimal penalty in memory usage. TinyVT employs a simple continuation mechanism to permit blocking wait, thus allowing split-phase operations within C control structures without relying on multiple stacks. Furthermore, it provides fine-grained scoping of variables shared between event handlers resulting in safer code and allowing for optimizations in compile-time memory allocation. TinyVT source code is mapped to nesC with a source-to-source translator, using synchronous communicating state machines as an intermediate representation.

1 Introduction

Most programming environments for wireless sensor nodes are based on one of the two dominating programming abstractions for networked embedded systems: event-driven or multi-threaded programming. In the event-driven paradigm, programs consist of a set of actions that are triggered by events from the environment or from other software components. Actions are implemented as event handlers: functions that perform a computation and then return to the caller. Event handlers run to completion without blocking, hence, they are never interrupted by other event handlers. This eliminates the need for locking, since event handlers are atomic with respect to each other. Furthermore, because of run-to-completion semantics, all event handlers can use a single shared stack.

In the multithreaded approach, execution units are separate threads with independent, linear control flow. Threads can block, yielding control to other threads that execute concurrently. Since the execution of threads is interleaved, data structures that are accessed by multiple threads may need locking. Each thread has its own stack and administrative data structures (thread state, stack pointer, etc.) resulting in memory usage overhead which may become prohibitive in resource-constrained systems.

Although the two abstractions were shown to be duals [1], there has been a lot of discussion about the advantages and drawbacks of both approaches in the literature [2][3][4]. Multithreading, especially preemptive multithreading, is commonly criticized for the nondeterministic interleaved execution of conceptually concurrent threads [5]. Various locking techniques are used to reduce (or eliminate) nondeterminism from multithreaded programs. Unfortunately, identifying critical sections, as well as choosing the appropriate lock implementations for the critical sections are error prone tasks. Suboptimal locking may lead to performance degradation, while omitting locks or using the wrong kind of locks result in bugs that are notoriously hard to find.

The most compelling advantage of multithreading is that the thread abstraction offers a natural way to express sequential program execution. Since threads can be suspended and resumed, blocking calls are supported: when a long-running operation is invoked, the thread is suspended until the operation completes and the results are available. The event-driven approach, in contrast, does not have this feature. Consequently, sequential execution involving multiple event handler invocation contexts is hard to express, and the corresponding event-driven code is hard to read.

The sensor network community is slightly biased toward the event-driven paradigm. The reason behind this tendency is twofold. First, the event-driven model reflects intrinsic properties of the domain: sensor nodes are driven by interaction with the environment in the sense that they react to changes in the environment, rather than being interactive or batch oriented. Second, the limited physical memory inhibits the use of per thread stacks, thus limiting the applicability of the multi-threaded approach. It is important to note here that Moore's law has an unorthodox interpretation here: it is applied toward reduced size and cost, rather than increase in capability, therefore, the amount of available physical resources is not expected to change as the technology advances.

The event-driven paradigm, nevertheless, has its caveats [2]. Real-time response to interrupts is not possible in traditional event-driven systems, since events cannot be preempted, thus interrupts must be stored and executed later. Relaxing this requirement would violate the atomicity of events, and could introduce race conditions necessitating locking. Events are required to complete quickly, because long-running computations can deteriorate the responsiveness of the system. To avoid this, complex CPU-intensive operations have to be split up into multiple event handlers. This constraint, however, hinders the portability of code that is not written in this event-aware style.

We have identified three issues that can have significant implications on the reliability and maintainability of event-driven code. First of all, unlike the thread abstraction, the event-driven paradigm does not offer linear control flow. The program execution is split up into actions that are executed in response to events.

It is often required, however, that an event triggers different actions depending on the program state. Traditional programming languages do not support dispatching different actions depending on *both* event type *and* program state. To tackle this issue, programs have to be implemented as state machines. Without explicit language support, these state machines are implemented in an unstructured, ad-hoc manner. As a result, the program code is often incomprehensible, error-prone and hard to debug. Second, sharing information between actions also lacks language support, and hence, programmers tend to use global variables, which is also error-prone and often suboptimal with respect to static memory usage. Third, since the event-driven paradigm does not allow blocking wait, complex operations must be implemented in a *split-phase* style: an operation request is a function that typically returns immediately, and the completion is signaled via a callback. This separation of request and completion, however, renders the use of split-phase operations impossible from within C control structures (such as `if`, `while`, etc.).

To address the above limitations, this paper introduces TinyVT, an extension of the nesC [6] language that provides a thread-like programming abstraction on top of the execution model of TinyOS [7]. The novelty of this approach is that threading is "compiled away:" programs that are expressed in a linear, thread-like fashion are compiled into event-driven nesC code. TinyVT has several important features that increase the expressiveness of the code and help improve the reliability of TinyOS components and applications:

Threads. The thread abstraction allows programs to be written in a linear fashion without sequencing event handler executions via explicit state machines. TinyVT threads are static in the sense that they are defined compile-time and they cannot be dynamically spawned. TinyVT threads are non-reentrant and stackless, thus very lightweight: only one byte is required to store the current state of the thread.

Blocking Wait. TinyVT employs a simple continuation mechanism, allowing threads to block on events. Blocking is also allowed within C control structures. Blocking on an event yields control to other threads or to the TinyOS scheduler, therefore, the execution of multiple concurrent threads is interleaved. Note that TinyVT does not require any scheduler other than the standard one provided by TinyOS.

Automatic Variable Allocation. TinyVT offers C-style scoping and automatic allocation of variables local to a thread, eliminating the need for global variables for information sharing between related actions. TinyVT does not require per thread stacks: local variables within a scope that includes at least one yield point (i.e. blocking wait) are statically allocated, while automatic local variables that are not shared between event handlers are allocated on the (single, shared) stack. To optimize memory usage, statically allocated shared variables use the same memory area if their lifetimes do not overlap.

Synergy with NesC. Since TinyVT is an extension of the nesC language, mixing nesC code and threading code is allowed. The TinyVT compiler, a source-to-source translator that maps TinyVT code to the nesC language, only processes the threading code within the modules, leaving any event-based nesC code unmodified. The generated code is subject to static analysis (data-race detection) and optimization by the nesC compiler.

Static Program Analysis. TinyVT code, due to the static nature of the language, lends itself to program analysis. The TinyVT compiler decomposes the threading code into communicating finite state machines. This FSM-style decomposition allows for static checking of safety properties, such as deadlock-freeness.

Run-Time Safety. Depending on the program state, some input events may not always be enabled. While nesC does not explicitly offer language support to manage component state, TinyVT does address this issue: the TinyVT compiler knows which events are enabled at a given point of program execution. If an unexpected event occurs, an exception handler is invoked. If no exception handler is specified, the execution of the program is halted to avoid nondeterministic behavior.

The rest of the paper is structured as follows. Section 2 provides a brief overview of TinyOS and nesC, introducing the terminology used in subsequent sections and setting the context for the rest of the paper. Then, we present the motivation of our work showing that the inherent complexity of event-driven software is difficult to manage. In section 4 we introduce the syntax of TinyVT and demonstrate the expressiveness of the threading abstraction through an example. Section 5 discusses how threading code is mapped to the event-based execution model of TinyOS. Since there is a large semantic gap between the levels of abstraction, the mapping is implemented in two phases. We describe an intermediate representation of component-based event-driven programs using synchronous communicating state machines as a vehicle, and explain how it maps to nesC. Then, we describe the challenges of translating threading code to the intermediate representation. Finally, we discuss the advantages, as well as the limitations of our approach, comparing it to related work in the field of sensor network operating systems.

2 TinyOS and the NesC Language

This section describes TinyOS [7], a representative event-driven operating system for networked embedded systems, and its implementation language, nesC [6]. NesC and TinyOS have been adopted by many research groups worldwide. TinyOS has been ported to a dozen hardware platforms, and a rich collection of software components is available. TinyOS and nesC provide low-level access to hardware, a flexible, event-based concurrency model, and a component-based architecture promoting modularization and reuse.

2.1 Concurrency Model

Though TinyOS is and event-based operating system, its concurrency model differs from that of the traditional event-driven paradigm. Based on the observation that data processing and event arrival from the environment are intrinsically concurrent activities in sensor nodes, TinyOS models concurrency with *tasks* and *events*. A task represents deferred computation that runs to completion without being interrupted by other tasks. Events represent interrupt contexts: they can preempt tasks as well as other events. Tasks are scheduled by a FIFO scheduler. Posting tasks is allowed from both task and event contexts.

Obviously, the two levels of parallelism in TinyOS may result in race conditions, and thus, variables that are accessed from interrupt context may need locking. However, the static nature of the nesC language (i.e. no function pointers or dynamic memory allocation is allowed) allows for compile-time data-race detection providing an adequate solution to this issue.

2.2 Component-Oriented Architecture

TinyOS provides a set of reusable system *components*, with well-defined, bidirectional interfaces. Common OS services are factored out into software components, which allows applications to include only those services that are needed In fact, the core OS requires just a few hundred bytes of RAM. There are two kinds of components in nesC: *modules* and *configurations*. Modules contain executable code, while configurations define composition by specifying encapsulated components and static bindings between them. A nesC application is defined as a top-level configuration.

Bidirectional *interfaces* provide a means to define a set of related (possibly split-phase) operations. Interfaces declare *commands* and *events*, both of which are essentially function declarations. A component *providing* an interface must provide the implementations of the interface's commands, and may signal events through the interface. A component that *uses* an interface can call the commands, and must implement callback functions for the events.

3 Managing the Complexity in Event-Driven Software Development

To demonstrate the inherent complexity of event-oriented programming, we present two examples. The first example, a packet-level I^2C driver, shows that managing control flow manually can be challenging, even in simple applications. The second example, a matrix multiplication, suggests that it is nontrivial to port code that implements a long-running computation, such as encryption key generation or data compression, to an event-driven platform.

3.1 Example: I2C Packet Level Interface

Let us consider the implementation of a packet-level interface for the I2C bus that operates above the byte-oriented hardware interface. The corresponding module should provide split-phase operations to write a packet to, and to read a packet from the bus. We only present packet sending; reading a packet works analogously.

The hardware interface provides the following operations. Starting of the send operation is requested with the `sendStart` command, to which the hardware responds with a `sendStartDone` event. Sending a byte is also implemented in a split-phase style: the hardware signals the completion of the write command with a `writeDone` event. After all the bytes are written, the bus has to be relinquished with a `sendEnd` command, the completion of which is acknowledged with the `sendEndDone` event.

The following pseudocode describes the procedure that writes a variable-length packet to the bus, using the byte-oriented hardware interface:

Algorithm 1. Pseudocode of the `writePacket` command in a packet-level I^2C interface

```
1: procedure I2CPacket.writePacket($length$, $data$)
2:     call I2C.sendStart
3:     wait for I2C.sendStartDone
4:     for $index = 0$ to $length$ do
5:         call I2C.write($data[index]$)
6:         wait for I2C.writeDone
7:         $index = index + 1$
8:     end for
9:     call I2C.sendEnd
10:    wait for I2C.sendEndDone
11:    signal writePacketDone
12: end procedure
```

Expressing this behavior in a linear fashion, however, is not possible in an event-driven system. The code must be broken up into a `writePacket` command and three event handlers, and the control flow must be managed manually. Variables that are accessed from more than one event handlers (`length`, `data`, and `index`) must be global and statically allocated. Typically, manual control flow is implemented with a state machine: a global static variable stores the component state, while the transitions of the state machine are coded into the event handlers. Commonly, only a restricted subset of input events is allowed at a given point of execution. Because of this, actions in the event handlers must be protected against improper invocation patterns (e.g. `writePacket` can only be called again after the previous packet sending is finished).

Manual management of control flow can become particularly tedious and error-prone as the complexity of the task increases. Breaking up the code into event handlers inhibit the use of loops and conditionals with blocking wait. As a result, even a simple control flow that can be expressed linearly with a few nested loops, may result in very complex state machines. Moreover, the resulting event-driven code will most probably be suboptimal, unclear, hard to debug, and often incorrect.

Efficient allocation of variables that are shared between multiple event handlers is also a challenging task in the presence of resource constraints. Notice that variables associated with sending a packet, and variables used when reading a packet might never be used at the same time. In a thread-oriented programming model, such variables are created on the local stack, and destroyed when they go out of scope. A similar, manual stack management approach appears in some event-driven components: variables with non-overlapping lifetime can be placed into a union allocated in static memory, thus the component consumes no more static memory than the memory required by the maximal set of concurrently used variables. However, such optimizations can be extremely tedious when the component logic is complex.

3.2 Example: Matrix Multiplication

Long-running computations may deteriorate the responsiveness of event-driven systems, since events are atomic with respect to each other and cannot be preempted.

This problem also manifests itself in cooperative multi-threading, however, such systems commonly provide a yield operation, by which the running computation may relinquish control and let other threads execute. This, however, is not possible in an event-driven programming paradigm.

Consider the multiplication of two fairly large matrices, a computation that is prohibitive in an event-driven system that has to handle various other events (e.g. message routing) concurrently. The most straightforward solution to this problem is to break up the outermost loop of the matrix multiplication algorithm, and to manage the control flow with a state machine emulating the loop.

This workaround, although typically tedious, will always work. However, this has serious implications: since it is cumbersome to emulate yield in event-driven systems, existing code which is not structured in an event-aware fashion can be extremely complex to port. This applies to computationally intensive algorithms, such as encryption key generation or data compression.

4 The TinyVT Language

In this section we overview the syntax and operational semantics of TinyVT, and through an example, we illustrate how TinyVT simplifies the development of event-driven applications.

4.1 Language Constructs

TinyVT extends the nesC language with two basic construct: *threads* and blocking *await* statements. Threads describe sequential blocks of computation with independent, linear control flow. The execution of concurrent threads is interleaved. A thread may pass control to another thread by signaling an event on which the other thread blocks, or, in TinyVT terminology, upon which the other thread *awaits*. Blocking wait can be expressed with the *await* statement. The await statement specifies one or more events, with the corresponding event handling code inlined, on which the thread blocks. Await has *OR* semantics: if the thread blocks on multiple events, the occurrence of any one of them resumes the execution of the thread. Thread execution continues with the execution of the body of the event handler of the triggering event, and the thread keeps running the code following the event handler till the next blocking statement is reached.

Event handlers cannot contain blocking code. The body of the event handler must be a valid nesC compound statement (i.e. function body), with the exception that either *dreturn* or *ireturn* should be used instead of the standard C *return* statement. Deferred return, or *dreturn*, means that after the execution of the event handler finishes, the control is *not* passed immediately back to the caller, instead, the thread continues running until the next blocking statement. In contrast, immediate return, or *ireturn*, returns to the caller "almost immediately": before actually returning, it posts a task which when scheduled, resumes the execution of the thread with the code following the await statement in which the event handler resides. Hence, *ireturn* defines an implicit yield point after the await statement. Using both deferred and immediate return is allowed within the same event handler. For clarity, it is required that functions with no return value should explicitly specify their return style with deferred or immediate return statement(s).

Threads may contain *yield* statements that explicitly transfer the control back to the caller of the event that invoked the currently running computation. Yield is syntactic sugar: it is essentially equivalent to posting a task and then blocking on it.

Threads react to events from the environment by executing a series of actions until the execution reaches a yield point (*await*, *yield* or *ireturn* statement). With each accepted event, the execution of the thread progresses. In fact, TinyVT threads can be thought of as state machines that are described in a linear, thread-like fashion, where the states are associated with yield points, and actions are associated with the code between them.

Since actions run in the execution contexts of the triggering events, there is no dedicated execution context associated with a TinyVT thread. In traditional multi-threading, there is a stack associated with each thread. In contrast, TinyVT threads use a common, shared stack, which is unrolled every time the thread blocks. Because of this, variables with lifetime spanning multiple actions must be statically allocated. TinyVT shields this from the programmer: automatic variables are allowed within threads and allocated in static memory. Because of the static nature of the language, call graphs are known compile time, thus further optimizations are possible: automatic variables that cannot be active concurrently are allocated at the same memory area.

TinyVT threads are not reentrant. A thread reacts an event only if it explicitly blocks on it. If an event is received when the thread is executing an action, or, if the thread is blocked, but the input event is not among the ones the thread is awaiting, an exception occurs. The default behavior on an exception is to halt the execution of the program, in order to prevent nondeterministic behavior. However, the programmer can implement a custom exception handler per event type, and may choose to recover from the error. This behavior may seem as a restriction, but it is in face equivalent to the behavior of event-driven systems: without extra logic, a program cannot handle a new message, for example, while the previous one is still being processed.

Structurally, threads reside within nesC modules. One module may contain multiple threads. Threads can access the local state of the module (i.e. global variables), can invoke functions at the module scope as well as through the module's interfaces (in nesC terminology: *call commands* and *signal events*), and can react to function calls through the interfaces. Threads are static in the sense that they are known at compile time, and cannot be dynamically spawned. Hence, threads are statically instantiated when the application starts. Instead of transferring control to the threads immediately after the application is bootstrapped, we require that the first statement in a thread be an await statement. This way, modules containing threading code are not bound to using a TinyVT specific interface. As a result, the fact that a module contains threading code is not visible from outside: they can be used in component specifications equivalently to standard nesC modules. This limitation reflects the event driven programming practice that components do not start executing immediately at boot-up time, instead, they initialize in response to an *init* command.

4.2 Example

We illustrate the expressiveness of TinyVT by rewriting the I2C packet-level interface example using the thread abstraction.

In the idle state, i.e. when no client request is being processed, the thread blocks on the `writePacket` command. If a client request comes in, the inlined implementation of the command is executed, requesting access to the bus by calling the `sendStart` command. The thread blocks as the next await statement is reached. The occurrence of the `sendStartDone` event, signaled by the byte-level hardware interface, resumes the thread execution. Since the corresponding event handler returns with a deferred return statement, the return value will be saved in an automatic temporary variable, and the same event context will continue running the code up to the next blocking statement. That is, the initialization of the index variable, the evaluation of the loop condition, as well as writing the first byte to the I2C bus will take place before the thread blocks again.

Notice that the execution of the thread is driven by the incoming events. TinyVT generalizes the concept of events to nesC commands, TinyOS tasks, as well as to local functions: a thread can block on any of these. Mixing multiple event types in one await statement is also allowed.

TinyVT supports run-time safety checking through exception handlers. For example, if a `writePacket` call comes in from the client while there is another packet being processed, the control is passed to an exception handler. The default behavior of the exception handler is to halt the execution of the application. However, the programmer may define custom exception handling code. In this example, we can assume that the hardware adheres to the contract defined by the I2C interface, but we need to prepare for handling client calls at any time.

```
uint8_t *packet_data; uint8_t packet_length; uint8_t index;
await result_t command I2CPacket.writePacket(
              uint8_t length, uint8_t* data)
{
    packet_data = data;
    packet_length = length;
    call I2C.sendStart();
    dreturn SUCCESS;
}
await result_t event I2CP.sendStartDone() {
    dreturn SUCCESS;
}
for(index=0; index<packet_length; ++index) {
    call I2C.write(packet_data[index]);
    await result_t event I2C.writeDone()
    {
        dreturn SUCCESS;
    }
}
call I2C.sendEnd();
await result_t event I2C.sendEndDone() {
    dreturn SUCCESS;
}
signal I2CPacket.writePacketDone(SUCCESS);
```

Fig. 1. Excerpt from the packet-level I2C interface module implemented with TinyVT threads. Notice how this code resembles the pseudocode presented in Alg. 1.

Therefore, the thread has to be protected with an exception handler, which is a nesC function definition with the *unexpected* qualifier:

```
unexpected result_t command I2CPacket.writePacket(
              uint8_t length, uint8_t* data)
{
    return FAIL;
}
```

Fig. 2. Exception handler in TinyVT

5 Mapping of the Threading Abstraction to Event-Driven Code

Although TinyVT offers a thread-like programming abstraction capable of expressing linear control flow, it is important to note that TinyVT threads are very much *unlike* threads in the traditional sense: there is no explicit execution context associated with a thread. Furthermore, the resulting event-driven code requires no multi-threading OS support nor does it introduce dependence upon a threading library. TinyVT threads are *virtual* in the sense that they only exist as an abstraction to express event-driven computation in a sequential fashion, and are transformed into (non-sequential) event-driven code by the TinyVT compiler.

While in traditional threading, context management and continuation support comes from the operating system or from the hardware essentially for free, TinyVT has to address these issues at the compiler level. Since there is a significant semantic gap between the thread abstraction and event driven-code, we introduce an intermediate representation, based on synchronous communicating state machines, that establishes an execution model on top of an event-driven system, and serves as a compilation target for the TinyVT compiler.

5.1 Operational Semantics of the Intermediate Representation

The execution model of TinyVT is based on tightly coupled, synchronous communicating state machines (SCSM), providing an expressive vehicle to capture the structure, the control flow, the state, and the communication patterns of event-driven software components. Although the SCSM representation is influenced by communicating finite state machines [8], instead of being a modeling language with well-defined denotational semantics, it primarily focuses on executability of the model rather than providing a mathematically sound foundation for creating correct-by-construction systems. As SCSMs are used exclusively as an intermediate representation, we do not define a concrete syntax for the language here.

An SCSM is defined by a finite set of *states*, *input events*, and *transitions* that map states and events to other states. Transitions are associated with *actions*, which are units of computation defined in the host language. To reduce state space, SCSM allows for the definition of *state variables*, which, depending on their scope, may be accessed from multiple actions. Actions typically read and update shared state variables, and generate *output events*.

It is valid to omit the triggering event from the definition of a transition. If the event is omitted, the transition fires immediately after the source state of the transition is entered. Transitions without events are allowed to have *guard* conditions, which are predicates over the state variables and are evaluated when the source state of the transition is reached. A transition can fire only if the predicate holds. SCSM does not allow specifying both a guard and a triggering event for a transition. Furthermore, mixing event-triggered and guarded output transitions from the same state is also not allowed: for any given state in a well-formed SCSM, either all or none of the output transitions are defined with events. These limitations partition the states into two sets: *blocking states*, in which the state machine is waiting for an external event, and *transitory states*, that are immediately exited after being entered.

SCSMs are deterministic: if a state has multiple out-transitions the same event cannot be assigned to more than one transition. Alternatively, for transitory states, the guards corresponding to the transition must be mutually exclusive.

Unlike traditional FSM models, SCSM does not assume that transitions are instantaneous. Therefore, input events are disabled during the execution of actions, and are re-enabled only after the action completes and the target state is reached. If an input event occurs when the state machine cannot handle it (referred to as an *exception*), the state machine immediately transitions to a (terminal) error state.

SCSMs communicate with their environment through input and output events. Communication between state machines is synchronous: if an action in machine **A** generates an output event which is accepted by machine **B**, **B** starts executing, and the action in **A** that generated the event blocks until **B** relinquishes control. That is, control flow is synchronously passed between communicating state machines. Multiple state machines may react to the same event. The execution of the corresponding event handlers is serialized, but their execution sequence is undefined.

The SCSM language supports hierarchical composition. The composition of state machines **A** and **B** is defined as a SCSM, such that the state set of the composite state machine is the Cartesian product of the states of **A** and **B**, and the input and output events of the constituent state machines are matched by name. Composition allows for event renaming, thus supporting arbitrary associations of input and output events, including fan-in and fan-out. Furthermore, it allows for event hiding, forbidding the propagation of the hidden input or output events over the composite state machine's boundary.

5.2 Mapping the State Machine Model to Event-Driven Code

The SCSM representation is conceptually an extension of the event-driven execution model of TinyOS, with a structure that resembles that of component-oriented nesC programs.

The nave way of implementing an SCSM in nesC is as follows. The state is stored in a global integer variable. Actions are implemented as functions at the nesC module scope. The transition system, i.e. the control logic that maps events and state to actions, is factored out to a scheduler function. When an input event occurs, to which the state machine reacts, it is handled by a generated event handler that calls the scheduler with the event type as a parameter. The scheduler decides which action to call depending on the event type and the current state. After the event handler completes, the scheduler updates the state. If the new state is a transitory state, the scheduler evaluates the guard conditions and invokes an action accordingly, again, updating the state when the action completes. This is iterated until a blocking state is reached. After entering a blocking state, the scheduler returns control to the generated event handler, which then returns to its caller.

The mapping of SCSM to nesC, as described above, is simple, it has limitations. Events commonly have formal parameters, as well as a return value. Passing parameters and return values between the generated event handlers and the actions is cumbersome, because every call has to go through the scheduler function, which has a fixed signature. Instead of trying to find a workaround for this issue (e.g. packing parameters into a variable length untyped array), we factored out the scheduler functionality into the generated event handlers and into the actions.

The generated event handler does the dispatching depending on the value of the state variable. Since the signature of the generated event handler and the corresponding actions are identical, the issue of parameter passing is eliminated. After the action returns, the generated event handler saves the returned value into a temporary local variable, and updates the state. If the resulting state is transitory, the actions and the state updates are executed iteratively, until a blocking state is reached. Then the generated event handler returns with the return value that was saved in the temporary variable.

5.3 Transforming Threading Code to the Intermediate Representation

TinyVT threads that do not declare automatic variables nor use branching or loops (i.e. C control constructs such as `if`, `while`, etc.) can easily be translated into SCSM. Await statements mark blocking states, the inlined event handlers are the corresponding actions. Immediate return statements in the event handlers are translated into three statements: setting a global thread-specific return type flag to `IRETURN`, posting the thread-specific continuation task, and a standard C return statement with the given return value. In the case of deferred returns, the return type flag is set to `DRETURN`, no continuation task is posted, and the standard C return statement is generated. The next state for all event handlers is a transitional state with two output transitions guarded by the return type flag: on `IRETURN`, the next state is a blocking state, awaiting the thread specific continuation task, which when executed, transitions the state machine into a transient join state. On `DRETURN`, the next state is the join state. Code following the await statement but before the next blocking statement is wrapped into an action, which is assigned to a transition from the join state to the blocking state marked by the next blocking statement.

C control structures that contain blocking statements are implemented with transient states branching based on the evaluated condition expression. We explain the translation of the `while` statement; other control structures (`for`, `if`, etc.) are implemented similarly. The `while` statement is translated to a transitory initial state that unconditionally transitions to a branching state, executing an action that evaluates the loop condition. The branching state is a transitory state that transitions to the transitory join state with an empty action if the loop condition evaluated to `FALSE`. On `TRUE`, the next state is the initial state of the state machine that corresponds to the body of the while loop. The final state of the enclosed state machine is linked to the initial state of the while statement with an empty action. The body of the while loop is processed recursively: the corresponding SCSM is built similarly as described a paragraph earlier, resolving C control structures if needed.

It is important to note that not all C control structures need to be converted to SCSM representation. If a control structure does not include any blocking code, it can be treated as a *primitive* statement, which is allowed within actions.

The compiler can decide if a control structure has blocking code by post-order traversing the abstract syntax tree and marking the nodes of statements with blocking descendants.

Automatic local variables declared within primitives can be allocated on the shared stack, since their lifetime is limited to a compound statement that will execute within one event context. However, if the scope of the variable is a compound statement that contains blocking code, the variable has to be allocated in static memory, since the shared stack is unrolled every time the thread blocks.

It is easy to see that compiler-managed variables with non-overlapping scopes can be allocated at the same static memory address. The compiler solves this by creating a `struct` for each compound statement, which contains the local variables, and a `union` containing the `struct`-s of non-overlapping child scopes, recursively.

6 Discussion and Future Work

We believe that the execution model of TinyOS coupled with the nesC programming model is a good level of abstraction for developing sensor node applications. In the presence of severe resource constraints, language support for low-level interfacing with the hardware is imperative. Although nesC provides a sophisticated component-oriented programming model that helps manage the structural complexity of sensor node applications, the inherent complexity of event-driven control flow may persist at the module level.

The virtual threading that TinyVT provides helps mitigate this complexity. It must be emphasized, however, that the goal of TinyVT is *not* to provide an abstraction that shields the event-driven nature of the OS from the programmer. Instead, it serves as a tool that improves code readability, reduces development time, yet retains the low-level hardware access and flexible control of resources provided by the host language. Indeed, it is imperative that the programmer be aware that a TinyVT thread is just a virtual thread, and have an understanding of the compilation process.

TinyVT is not a silver bullet. It is widely known that not all patterns of sequential control flow can be expressed in a thread-like fashion. Analogously, the behavior of some nesC modules is cumbersome, if not impossible, to express in TinyVT. This particularly holds for components operating on top of a hardware presentation layer with nested interrupts. Since TinyVT threads are not reentrant, the programmer has to assure that asynchronous events are handled in a timely manner, alternately, unexpected events have to be handled adequately. Nevertheless, the programmer can always fall back to using plain event-driven nesC code in such cases, and write TinyVT modules only when it is convenient.

Our compiler prototype, though it processes the whole application to resolve symbols and wirings, considers only the scope of the shared variables when optimizing memory allocation, and does not detect if variables in different threads (or modules) can be allocated to the same memory address. That is, the optimization is local to a thread. Extending this functionality with whole-program analysis to facilitate global optimization is subject of further research.

Currently, we do not support all nesC features in TinyVT threads. For example, `goto` is not allowed, and `switch` statements containing blocking code are also not handled. It is primarily because the C standard is very permissive regarding labels, and the compilation of such code can be complicated. We consider eliminating these limitations in the future only if there is a demand for the currently unsupported language features.

Another exciting future direction is extending the compiler with a more thorough interface compatibility checking, based on the communication patterns exhibited/supported by the components through their interfaces. Since TinyVT threads express computation in a linear fashion, the communication patterns of modules are encoded in the control flow. Though TinyVT actions allow for data-dependent behavior, we suspect that some errors, such as violations initialize-before-use constraints, might be able to be detected via static analysis.

7 Related Work

Contiki [9] is a multitasking operating system for memory-constrained devices built around a small event-driven kernel. Unlike traditional operating systems, the Contiki kernel does not provide explicit support for multithreading. Instead, multithreading is implemented as an external library, which is linked into the application only if explicitly needed. Since each thread requires its own stack, traditional multithreading is expensive on memory-constrained platforms. As an alternative, Contiki promotes the use of protothreads [10]. Protothreads achieve threading without per thread stacks using a lightweight continuation mechanism, called local continuations, implemented as a set of C macros. The use of continuations is limited to a C function block, consequently, protothreads cannot span multiple functions. Protothreads in Contiki are similar to threads in TinyVT in that both approaches provide a threading context on top of an event-driven execution model. Protothreads take an opportunistic approach by exploiting esoteric or non-standard features of the C language, while in the TinyVT language a thread is a first class object with explicit compiler support. In contrast to TinyVT, automatic local variables in a protothread are not preserved when the protothread blocks, which can result in potentially unsafe code.

MANTIS [11] is a multithreaded operating system for wireless sensors built around classical concepts, such as preemptive scheduling with time slicing, kernel-level support for synchronization, etc. MANTIS provides a familiar API which is easy to use, making it particularly suitable for experimentation with new algorithms or rapid prototyping of sensor network applications. However, because of the need for per thread stacks, traditional multithreading is costly: MANTIS trades RAM usage for flexibility and ease of use.

TinyOS [7] is probably the most popular operating system in the wireless sensor networks domain. In TinyOS, the event-driven model was chosen over the multithreaded approach due to the memory overhead of the threads. TinyOS defines two kinds of execution contexts: tasks and events. Tasks are scheduled by a FIFO scheduler, have run-to-completion semantics, and are atomic with respect to other tasks. TinyOS models interrupt service requests as asynchronous events: events can interrupt tasks, as well as other asynchronous computations. This duality provides a flexible concurrency model, and easy interfacing with the hardware, however, it can introduce race conditions and may necessitate locking.

nesC [6], the implementation language of TinyOS addresses this issue by providing language support for atomic sections and by limiting the use of potentially "harmful" C language features, such as function pointers and dynamic memory allocation. nesC is a "static" language in the sense that program structure, including the static call graph and statically allocated variables, are known compile time, allowing for whole-program analysis and compile-time data-race detection. TinyVT inherits these features from nesC, while extending the language with support for threading and blocking wait. TinyVT overcomes the problem that complex operations have to be implemented using explicit state machines in nesC, hence, improving code maintainability and safety. nesC has a component oriented design that allows partitioning the applications, which is largely orthogonal to the execution model of TinyOS. This gives flexibility to the programmer and promotes reuse.

TinyGALS [12] defines a globally asynchronous and locally synchronous a programming model for event-driven systems. Software components are composed locally through synchronous method calls to form modules, modules communicate through asynchronous message passing. Local synchrony within a module refers to the flow of control being instantaneously transferred from caller to callee, while asynchrony means that the control flow between modules is serialized through the use of FIFO queues. However, if modules are decoupled through message passing, sharing global state asynchronously would incur performance penalties. To tackle this, the TinyGALS programming model defines guarded synchronous variables that are read synchronously and updated asynchronously.

The galsC [13] language, an extension of nesC, provides high-level construct, such as ports and message queues, to express TinyGALS concepts. TinyGALS/galsC and our approach attack the same substantial problem, namely that managing concurrency with the event-driven paradigm lacks explicit language support. TinyGALS ensures safety through model semantics. In contrast, TinyVT promotes static analysis and runtime safety checking instead. While in TinyGALS modules are decoupled through message passing, and synchronous control flow is limited to the module scope, TinyVT does not impose limitations on the allowable communication styles. We believe that our approach gives more flexibility to the programmer with respect to choosing the right structural decomposition for a problem, whereas galsC could impose limitations on the program structure. For example, control flow from an interrupt context cannot propagate outside the module: hence, all tasks that are timing critical must be implemented within the module.

SOS [14] is a general-purpose operating system for sensor nodes with an event-driven kernel and dynamically loadable modules. SOS strictly adheres to the event-driven paradigm: events are atomic with respect to each other. To handle interrupts in a timely manner without operating in an interrupt context, the SOS kernel uses priority queues to schedule the serialized execution of events. Since interrupt contexts do not propagate into application code, applications can fully leverage the benefits of the atomicity assumption. SOS, similarly to TinyOS, would be an ideal compilation target for TinyVT.

The Object State Model (OSM) [15] employs attributed state machines to express event-based program behavior. The application of FSM concepts is a natural choice for the domain: actions are executed depending on the input event and the actual state, whereas imperative languages, such as C, lack explicit support to associate actions with both events and program state. OSM specification is translated to Esterel [16], a synchronous language, which then can be compiled into efficient C code by the Esterel compiler. The most significant contribution of OSM, however, is that it offers efficient allocation of shared variables based on their lifetime making this approach particularly suitable for programming resource-constrained devices. TinyVT employs a similar approach to allocate automatic local variables. An important difference is that our language constructs do not allow explicit association of shared variables with states (since the state machine model is used only as an intermediate representation, and the concrete syntax is less expressive), hence OSM can achieve slightly better memory usage. However, our approach offers excellent code readability, while OSM should rather be used as a target for automatic code generation.

8 Conclusion

The novelty of this work is that it provides language support to describe event-based computation in a well structured, linear fashion without compromising the expressiveness of the implementation language. The event-driven execution model of TinyOS remains exposed to the TinyVT programmer, along with all the features of the nesC language from supporting component-oriented programming to compile time data-race detection.

The ”virtual thread” that TinyVT introduces is a simple language extension that provides a means to express linear control flow and blocking operations. Yet, these threads do not suffer from the problem of nondeterminacy which multithreading is commonly criticized for. First, TinyVT implements a variant of non-preemptive multithreading by sequencing the execution of atomic event handlers. Non-preemptive multithreading offers significantly more determinism and better analyzability than its preemptive counterpart. Second, the syntax of TinyVT ensures that the programmer is aware of the control flow between conceptually concurrent threads. Calls to split phase operations explicitly state which thread the control is passed to; similarly, the *await* statement explicitly specifies the thread which the control is received from. This stands in contrast to the approach of general-purpose multithreading, where control flow is governed by the scheduling policies of the operating system or a user-space threading library, and the programmer has no insight into inter-thread control flow (except for locking decisions).

The TinyVT compiler automates the tasks that programmers traditionally do manually. As the complexity of applications keeps growing even in the sensor network domain, such tasks are becoming hard to manage. However, the TinyVT compiler can easily cope with this complexity, and thus, produce better quality and more reliable code than an average programmer.

References

1. Laurer, H.C., Needham, R.M.: On the duality of operating system structures. SIGOPS Operating Systems Review **13** (1979) 3–19
2. v. Behren, R., Condit, J., Brewer, E.: Why events are a bad idea (for high-concurrency servers). HotOS IX (2003)
3. Lee, E.: What's ahead for embedded software? IEEE Computer (2000) 16–26
4. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., , Douceur, J.R.: Cooperative task management without manual stack management. Proceedings of the USENIX Annual Technical Conference (2002) 289–302
5. Lee, E.: The problem with threads. IEEE Computer (2006) 33–42
6. Gay, D., Levis, P., v. Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesc language: A holistic approach to networked embedded systems. SIGPLAN (2003)
7. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., , Pister, K.: System architecture directions for network sensors. Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX) (2000)
8. Brand, D., Zafiropulo, P.: On communicating finite state machines. Journal of the ACM **30** (1983) 323–242
9. Dunkels, A., Grnvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. EmNetSI (2004)
10. Dunkels, A., Schmidt, O., Voigt, T.: Using protothreads for sensor node programming. The Workshop on Real-World Wireless Sensor Networks (2005)
11. et al, H.A.: Mantis: system support for multimodal networks of in-situ sensors. WSNA (2003) 50–59
12. Cheong, E., Liebman, J., Liu, J., , Zhao, F.: Tinygals: A programming model for event-driven embedded systems. Proceedings of the 18th Annual ACM Symposium on Applied Computing (SAC'03) (2003)
13. Cheong, E., Liu, J.: galsc: a language for event-driven embedded systems. Proceedigs of Design, Automation and Test in Europe **2** (2005) 1050–1055
14. Han, C., Kumar, R., Shea, R., Kohler, E., Srivastava, M.: A dynamic operating system for sensor nodes. In Proceedings of the 3rd international Conference on Mobile Systems, Applications, and Services (2005) 163–176
15. Kasten, O., Rmer, K.: Beyond event handlers: Programming wireless sensors with attributed state machines. The Fourth International Conference on Information Processing in Sensor Networks (IPSN) (2005)
16. Berry, G., Gonthier, G.: The esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19** (1992) 87–152