COMPILER-ASSISTED CONCURRENCY ABSTRACTION FOR RESOURCE-CONSTRAINED EMBEDDED DEVICES

By

Janos Sallai

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

May, 2008

Nashville, Tennessee

Approved:

Professor Janos Sztipanovits

Professor Akos Ledeczi

Professor Xenofon Koutsoukos

Professor Sandeep Neema

Professor Miklos Maroti

Acknowledgements

I would like to thank my advisors, Professor Akos Ledeczi and Professor Janos Sztipanovits, for their guidance and support. I had the pleasure to working in the sensor networks research group at the Institute for Software Integrated Systems at Vanderbilt University, in a team that, under Akos's leadership, earned wide recognition over the last few years. I am grateful to have had the chance to be a member of this group and to contribute to its success.

The motivating examples and the initial problem formulation of my work came, in large part, from Professor Miklos Maroti. My special thanks go to Miklos for his inspiration, enthusiasm and encouragement.

I owe special thanks to the other two members of my committee, Professor Xenofon Koutsoukos and Professor Sandeep Neema, for their inspiring questions and suggestions regarding my work. Altogether, I could not have wished for a better Ph.D. committee.

The research in this work was sponsored by the NSF ITR program on Foundations on Hybrid and Embedded Software Systems. Support for my graduate studies was provided in part by the DARPA/IXO NEST program (F33615-01-C-1903), the ORISE program at the Oak Ridge National Laboratory, the United Technologies Research Center, Crossbow Technologies, Inc. and the NSF NeTS program.

Table of Contents

List of Figures

List of Abbreviations

ADT	Abstract Data Type
ASM	Abstract State Machines
AsmL	Abstract State Machine Language
AST	Abstract Syntax Tree
ASVM	Application Specific Virtual Machines
BNF	Backus-Naur Form
DPC	Deferred Procedure Call
FA	Finite Automaton
FIFO	First In First Out
FSMD	Finite State Machines with Datapath
HOL	Higher-Order Logic
IPC	Inter-process communication
LCFG	Local Control Flow Graph
MCU	Microcontroller Unit
MMU	Memory Management Unit
MOS	MANTIS Operating System
OS	Operating System
OSM	Object State Model
POSIX	Portable Operating System Interface
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SR	Synchronous Reactive
SRAM	Static Random-Access Memory
TA	TinyVT Automaton
VCC	Voltage at the Common Collector

VM	Virtual Machine
WSN	Wireless Sensor Network

Chapter I

Introduction

1.1 Wireless sensor networks

In the mid 1960's, Gordon A. Moore observed that the number of transistors on an integrated circuit for minimum component cost doubles every two years. While the semiconductor industry did indeed increase the complexity of products as Moore had predicted, it was only lately that a new, unorthodox way of perceiving the effects of development trends in microelectronics has emerged. Applying Moore's law toward reduced size and cost, rather than increase in capability, has opened a new perspective, fostering research in a new application domain: wireless sensor networks (WSN) are envisioned as large networks of small, inexpensive devices that communicate with low-power radios, are potentially ad-hoc deployed and suitable for unattended operation. Potential application scenarios range from precision agriculture [11] through seismic monitoring [39] to military surveillance [72, 51, 50, 77] – typically leveraging that the sensor nodes are deeply embedded in the environment.

Power and resource constraints, as well as application requirements inherent to the WSN domain may render traditional operating systems and the corresponding programming abstractions inadequate for sensor nodes. Contemporary mainstream operating systems and development tools rely on the assumption that system memory and storage are abundant, and that processing power and communication bandwidth are ever-increasing resources. In contrast, a typical sensor node [66, 24, 43] is running on batteries and is built around a microcontroller with only a few kilobytes of internal SRAM. Consequently, memory is a scarce resource, and computation, as well as communication, are expensive in terms of power consumption.

1.2 Requirements

While it is obvious that traditional operating systems are too heavyweight for resource constrained sensors, the requirements for operating systems and development tools for wireless nodes are, interestingly, very much analogous to those in general purpose computing. The intrinsic concurrency of applications, as well as the demand for development tools to offer productivity, program safety and reliability, mandate that operating system designers address challenges similar to those in general purpose computing, but in presence of constraints specific to the WSN domain. The scope of research in this field goes beyond operating systems concepts and their implementations. In fact, it is never the operating system (or runtime) alone that is used to attack the problem, but an entire toolchain, which may or may not include an OS , libraries, build tools (compilers, linkers, etc.), analysis tools, debugging and code distribution support, runtime services, monitoring or management software, and a general know-how encompassing patterns and best practices of the particular toolchain.

Clearly, sensor operating systems and the complementary development toolchains must be able to satisfy the functional requirements of WSN applications. At the same time, it must be achieved in the presence of resource constraints, while offering the developers productivity in terms of coding effort, maintainability, and program reliability.

1.2.1 Functionality

From the aspect of functionality, sensor nodes are often required to respond to external events in a time-critical manner, while running several concurrent tasks with best-effort semantics [44, 32, 8, 13, 33, 46, 75].

As an example, let us consider an acoustic shooter location application, where the sensor network acts as distributed microphone array to pinpoint shot sources [77, 72, 50]. This application requires that three tasks be running on a node concurrently. Individual wireless nodes are accepting event data through the I2C interface from the acoustic daughter board, timestamp them, and forward them to the sensor fusion node. When an event of interest occurs, nodes are also required to act as message relays in a multihop routing network. Notice that while messages are being forwarded, nodes keep listening for acoustic events. Concurrently, in order to relate event timestamps taken at different locations, nodes are running a distributed time synchronization algorithm. It builds on distributed leader election, and uses linear regression on received message timestamps to compensate for clock skews.

In this scenario, handling acoustic event data is time-critical: a jitter of one millisecond corresponds approximately to one foot of distance error, which decreases the precision of the localization algorithm. Multihop routing, on the other hand, is best-effort: the redundancy of the measurements allows for a certain degree of message loss. Time synchronization requires that its underlying timestamping primitives have low jitter, however, timing requirements of other aspects of the algorithm are relaxed.

1.2.2 Resource constraints

In the absence of significant hardware constraints, real-time operating systems meet the above functional requirements. However, when hardware resources are tight, existing solutions prove to be too heavyweight.

MCU speed and functionality is typically limited due the requirement that devices need to operate on battery with and anticipated application lifetime of several months. Memory constraints, on the other hand, are related to manufacturing costs. Even in low-end microcontrollers, the transistors implementing the integrated SRAM constitute more than three quarters of the total transistor count. In high-end embedded processors, such as the Intel XScale, this number is close to 90% [33].

A representative sensor platform, the Berkeley MICA2 mote, is equipped with an Atmel ATMega128L RISC microcontroller, with 4kB of on-chip SRAM and 128kB of flash, operating at 7.2MHz. Typical values for power consumption at 3V VCC are 8mA and 3.2mA when MCU is active and idle, respectively [71]. Using various power save modes, power consumption can be reduced to a couple of hundreds of microamperes.

The implications of the resource constraints are twofold. First, since both the operating system and the application must fit into a few kilobytes of RAM , traditional operating system services are cumbersome to implement. Particularly, multithreading becomes challenging, because each thread would require a separate stack. Second, power constraints dictate that program code be efficient, minimizing the time spent in active mode.

1.2.3 Development effort

Finding a balance between functionality, resource-awareness and the productivity a tool provides is a challenging problem. As a rule, the higher the level of abstraction that a toolchain provides, the harder it becomes to support these abstractions while meeting tight resource constraints.

Depending on the level of abstraction the operating system and development tools provide, the *programming effort* to create a WSN application can vary significantly. For instance, developing applications for a a bare event-driven operating system is hard: programmers need to concentrate on subtle details of error-prone tasks, such as manual management of control flow and stack. In contrast, macroprogramming techniques, virtual machines and scripting engines offer short development time by shielding the programmers from low-level details (typically at the price of reduced flexibility and increased resource usage).

Code reuse across projects or across different hardware platforms can significantly reduce development time. Examples of code reuse include libraries, frameworks [79, 58], component architectures [32] or virtual machines [54, 55, 47].

Safety and *reliability* are important factors in embedded system development. It is hard to recover sensor nodes from run-time errors. As a result, they are typically programmed to reboot when a critical fault is detected. Tracing down the cause of a fault, and debugging a deployed WSN in general, are yet unsolved challenges. Researchers, therefore, mostly focus on static analysis, along with development tools that provide correct-by-construction applications with respect to properties such as race conditions, memory ownership, stack overflow, component compositionality, etc.

Maintenance effort of sensor network applications is also an important factor to consider. The cost of migration from one hardware platform to another, finding and fixing software errors and modifying source code to enhance functionality heavily depends on the comprehensibility of the sources. While the level of abstraction the programming environment provides will obviously have a significant effect on maintenance costs, the intrinsics and constraints of the underlying execution model are equally important factors to consider. For instance, independently from the implementation language, locking is challenging in a multithreaded system [52]. Similarly, in event-driven systems, the lack of explicit language support to express linear control flow cripples the comprehensibility of event-driven programs [20]. Alternatively, from a coarse-grained perspective, maintainability (and also reliability) of an application can be improved through structure, which is typically enforced by the development toolchain [32], and patterns [31, 41], i.e. best practices for the given toolchain or application domain.

1.3 Research directions

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.

1.3.1 Multithreading

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 (assuming one physical processor), 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 [49], there has been a lot of discussion about the advantages and drawbacks of both approaches in the literature [76, 52, 2]. Multithreading, especially preemptive multithreading, is commonly criticized for the nondeterministic interleaved execution of conceptually concurrent threads [53]. 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. Threads offer a natural way to model and implement conceptually independent or loosely coupled services used within the same WSN application. Using threads forces the programmer to think of the application as a set of services, and promotes the *functional decomposition* of sensor node software.

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.

1.3.2 Event-driven programming

A typical event-driven program consists of an event *dispatcher* (also called *scheduler*) and a set of event handlers. Depending on the requirements, applications may include a large number of event handlers, and, thus, the implementation can become very complex. To mitigate complexity, state-of-the-art event-driven operating systems for wireless sensor nodes [40, 44], however, allow for modularization of event-driven code.

Within an application, event handlers can be partitioned such that those that belong to the implementation of the same conceptual service are enclosed in a module. This is a widely applied design principle in software engineering, commonly called *functional* or *horizontal decomposition*. The list of such modules may include sensing, multihop routing, timer subsystem, etc. With such horizontal modularization in place, the event-driven application is not a set of event handlers any more: it is structured as set of event-driven service modules, enclosing event handlers that implement the service logic. Horizontal structuring implies a high level of isolation between modules, preventing or discouraging direct service to service interactions. This allows for developing modules separately, and promotes reuse.

To further enhance reusability and portability of event-driven code, monolithic services can be subdivided into layers, where layers implement subservices at different levels of abstraction. This *vertical decomposition* has a number benefits. First, common functionality can be refactored from modules into a shared, low-level service (e.g. a dissemination and a data collection service can use the same packet forwarding engine). This, of course, requires that the low-level service have a well defined interface, capturing the description of the service at the abstraction level which the service provides.

This way, *stacks* of services can built, where each layer of the stack implements an *abstract service* relying on the concepts of the underlying abstraction level. All layers, except for the lowermost one, are decoupled from concrete platform features. For example, a timer stack can be built such that timers are built from alarms, alarms are built from counters, and counters are implemented by directly interfacing with the hardware. Service stacks are essential to enable portability across hardware platforms, and to allow for reuse even if the hardware-software boundary changes with the evolution of hardware.

An important benefit that comes from vertical decomposition is that a complex service is refactored to a stack consisting of abstract services that are simpler to implement, since they are also built on a set of abstractions that hide the underlying complexity. This greatly simplifies the implementation of individual services, and promotes reuse of code not only between applications, but also between different platforms.

Vertical layering in an event-driven application implies that, for performance reasons, there is direct communication between layers, without involving the event-driven dispatcher. Although the implementation of an abstract service is still just a collection of event handlers, they are not necessarily invoked directly by the dispatcher: interacting layers may invoke each other's event handlers. In a clean event-driven application, however, all invocation contexts, directly or indirectly, must originate from the dispatcher.

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.

1.4 Problem statement

I have identified an number of issues that can have significant implications on reliability and maintainability of event-driven code, and also on the coding effort required to develop event-driven applications.

- **Manual control flow management.** 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 — unaware of the underlying event-driven runtime — does not allow for defining control flow spanning multiple event invocations. Neither do they support executing different actions depending on *both* event type *and* program state, hence the logical ordering of event invocations is not captured.

  To tackle this issue, services have to be implemented as state machines, constituting essentially *manual control flow management*. Without explicit language support, these state machines are implemented as a set of event handlers, in an unstructured, ad-hoc manner. As a result, the program code is often incomprehensible, error-prone and hard to debug.

- **Manual stack management.** Sharing information between event handlers 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.

  Efficient allocation of variables that are shared between multiple event handlers is a challenging task in the presence of resource constraints. 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.

- **Yield emulation.** Long-running computations should be avoided in event-driven systems, because they deteriorate overall system responsiveness, since event handlers execute atomically 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. Also, since the loop is broken up into two event handlers, manual stack management is required for the loop's local variables.

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.

- **Non-blocking split-phase operations.** 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.).

  Manual management of control flow can become particularly tedious and error-prone as the complexity of the programs increase. 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.

1.5 Contributions of this dissertation

In this work, I present a **compiler-assisted concurrency abstraction** that allows for thread-like programming, but retains the benefits associated with an event-driven execution context. I introduce TinyVT, an extension to ANSI C with explicit language support for the thread abstraction, including blocking statements. This *virtual* threading abstraction allows for implementing services, that have logically linear control flow, as if they had their own, independent threads of execution. The abstraction of a thread is provided by the language: a compiler will translate TinyVT code — containing multiple, conceptually concurrent threads — to C (or nesC [32]) assuming a simple event-driven execution context (e.g. SOS [40] or TinyOS [44]), by identifying and sequencing primitive blocks of source code that contain no blocking statements.

A common drawback of event-driven systems is that the lifetime of a local variable is limited to the execution context of the event handler in which the variable is declared. Typically, C compilers allocate local variables with automatic storage duration on the stack. Since in event-driven systems, the stack is unrolled every time the event handler completes, the values of such variables are not preserved between consecutive event invocations.

Consequently, variables that are accessed from more than one event handlers must be global, allocated in static memory. To overcome this issue (termed manual stack management), I introduce a **compiler-managed memory allocation** technique that seamlessly provides C-style scoping and automatic allocation of variables local to a thread, eliminating the need for declaring global variables for information sharing between related actions.

Although its significance is often understated or not even recognized, the precise **specification of semantics** is essential to any programming language. Vague and informal semantics may result in incompatibilities between tools that are built on conflicting assumptions of the ambiguous, incomplete or nonexistent specification. Semantic ambiguities may lead to undesired software behavior.

I specify the operational semantics of TinyVT threads using the Abstract State Machine (ASM) formalism (formerly known as Evolving Algebras [34]), building on an existing formal semantics specification for C. The notion of a thread, defined as an independent unit of computation with conceptually linear control flow, is missing from C, since the C language is a legacy of an era in which multithreading had not yet existed. Therefore, I investigate the compositional semantics of TinyVT threads by mapping them to a finite automaton based representation, the semantics of which is given in the Abstract State Machine Language (AsmL) [37].

1.6 Organization

This dissertation is structured as follows. In Chapter 2, I set the context of my work by introducing the related work in systems research for resource-constrained platforms, highlighting the main research directions and introducing the research problem. After giving an overview of the approach in Section 2.4, along of the definition of TinyVT threads, a compiler-assisted concurrency abstraction for event-driven systems, I present a language extension of ANSI C in Section 2.5, and describe the compiler that translates code expressing conceptually concurrent threads to standard C code in Section 2.6.

In Chapter 3, I specify the semantics of the language extension, and investigate the compositional and interaction semantics of the model of computation defined by TinyVT threads. Following a brief literature review on approaches to formal semantics specification in general, and that of the C language in particular, in Section 3.7, I present the semantics of the TinyVT language extension by extending the work of Gurevich and Huggins on the formal semantics of ANSI C [35] (Section 3.6) using the Abstract State Machines (ASM) [34] approach. In Section 3.8, I investigate the compositional and interaction semantics of TinyVT threads by mapping such systems to a finite automata based representation, the structural and behavioral semantics of which I specify formally in the Abstract State Machine Language (AsmL) [37].

Finally, in Chapter 4, I discuss the advantages, as well as the limitations of the TinyVT approach, highlighting points of possible improvement and future research directions.

Chapter II

Compiler-assisted threading

2.1 Background and related work

Operating systems, programming models, and development tools for wireless sensor nodes have been a very active research area in the recent years. In this section, I present a detailed review on the event-driven and the multithreaded approach to sensor node programming, and also highlight some alternative research directions.

2.1.1 Event-driven model

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 are executed in a serialized manner: once an event handler is invoked, it cannot be preempted and must run to completion. As a result, event handlers are atomic with respect to each other, hence the execution model is free from race conditions. An important implication of the lack of data races is that there is no need for locking in the event-driven programming model.

Runtime support for event-driven systems is fairly simple. A central component of the runtime is the *scheduler* (also called *dispatcher*). The scheduler, running in a loop, maintains a queue of events that have occurred (either externally to the system, or created internally by event handlers). While the queue is not empty, the scheduler removes an event from the queue based on the scheduling policy (which can range from a simple FIFO to sophisticated priority queues), and invokes the corresponding event handler. The event handler runs to completion, running the computation associated with its triggering event. Furthermore, it may create new events and place them on the schedulers queue. After the event handler returns, the scheduler dispatches the handler corresponding to the event that is at the head of the queue.

Implementing an event-driven system is cheap in terms of memory usage. Events can be modeled as function pointers to the corresponding event handlers, which, when stored in the scheduler queue, do not consume more than a couple of bytes each. An important advantage of the event-driven model over multithreading is that it can be implemented using a single stack that is shared between the scheduler and all event handlers. When an event handler is invoked, it can safely use the stack (for storing function return addresses, registers, automatic local variables), because it is guaranteed not to be interrupted by other event handlers. When the handler completes, the stack is unwound, and the next scheduled handler can take it over. Also, it is possible to precisely estimate the stack usage of an event-driven system with static analysis: the maximum stack usage of the whole system is the maximum of the stack usage of the individual event handlers. This allows system programmers to prevent stack overflows while avoiding overallocation of stack (such as in [68, 13]), which results in better RAM usage.

At any time, only one event handler can be active in the system. The active event handler must be completely executed before the next one can be scheduled, therefore, every event handler depends on the previous one to terminate in time. Long running handlers can "jam" the scheduler queue and increase the overall system latency. This, however, can be avoided by breaking up long tasks into smaller pieces, each of which, on completion, trigger the next piece with application-specific events, in a sequential manner.

In embedded systems, external events often manifest themselves as interrupt request. Serving interrupts, however, violates the atomicity assumption of the event handlers, and thus might undermine basic assumptions of the event-driven model. When an interrupt occurs, the currently executing handler is stopped and the control is passed to the interrupt handler. The interrupt handler can use the shared stack to store its return address, save registers and allocate variables. Typically, what the interrupt handler does is that it serializes the triggering hardware event by placing an event on the event queue. Then it unrolls the stack, returns, and the execution of the interrupted handler is resumed.

Letting interrupts interfere with the event-driven execution contexts can be a source of several problems. First, if there are variables which are shared between interrupt handlers and event handlers, race conditions may occur. In such systems, appropriate locking is required to avoid data races. Locking, in its simplest form, can be achieved by temporarily (and selectively) disabling interrupts. Second, execution times of event handlers become nondeterministic, as they can be interrupted arbitrarily. To ensure strict timing guarantees, interrupts typically need to be disabled. This, however, can lead to decreased responsiveness and overall throughput of the system. Third, stack usage of the system is affected by interrupt handling strategies as well. If interrupts are allowed to preempt each other, the stack can go arbitrarily. That is, maximum stack size becomes unpredictable: freedom from potential stack overflows cannot be guaranteed.

SOS

SOS [40] is lightweight operating system for wireless senor nodes built around the concepts of general purpose operating systems. Namely, SOS provides separation of kernel and user space, borrowing ideas from previous work on microkernels, such as Exokernel [26] or the Mach kernel [67].

The kernel is lean and modular: it provides only a minimal hardware abstraction for applications. Importantly, it deserializes interrupts, assuring that atomicity of userspace event handlers with respect to each other is retained. As a result, SOS provides a clean event-driven execution environment. This has many advantages, e.g. there are no data race conditions in user modules, there is no need for locking, and the whole system can be implemented to use a single stack.

However, timely response to interrupts is only possible in kernel drivers. As a result, the implementation of an application that requires strict timing has to split into kernel code and user modules, where the former might prove to be a highly nontrivial task.

TinyOS and the nesC language

TinyOS [44] 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. However, the execution model of TinyOS differs from that of clean event-driven systems. 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. In contrast to SOS, TinyOS does not have clear kernel/user mode separation. Interrupt contexts are not necessarily serialized in device drivers: the corresponding asynchronous event can propagate even to application-level modules. This duality provides a flexible concurrency model, and easy interfacing with the hardware, however, it can introduce race conditions and may necessitate locking.

nesC [32], 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.

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.

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 is captured in nesC's bidirectional interfaces. 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.

TinyGALS and galsC

TinyGALS [15] 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, and 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 [16] language attacks a substantial problem of event-driven programming, namely that managing concurrency with the event-driven paradigm lacks explicit language support. It is an extension of nesC that provides high-level constructs, such as ports and message queues, to express TinyGALS concepts. TinyGALS ensures safety through model semantics. As a tradeoff, galsC could impose limitations on the program structure. TinyGALS modules are decoupled through message passing, and synchronous control flow is limited to the module scope. As a result, control flow from an interrupt context cannot propagate outside the module: hence, all tasks that are timing critical must be implemented within the module. This is the price paid for containing potential data races within the module implementing the interrupt handler.

2.1.2 Multithreading

In the multithreaded approach, the units of execution 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 [49], there has been a lot of discussion about the advantages and drawbacks of both approaches in the literature [76] [52] [2]. Multithreading, especially preemptive multithreading, is commonly criticized for the nondeterministic interleaved execution of conceptually concurrent threads [53]. 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 conceptually concurrent threads of execution, each with independent, linear control flow. 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.

Contiki and Protothreads

Contiki [22] is an event-driven operating system for memory-constrained devices. Contiki is built using Protothreads [23], a programming abstraction that makes it possible to write eventdriven programs in a thread-like style with minimal memory overhead.

Protothreads are not threads in the traditional sense: protothreads are a non-obvious, albeit standard-compliant C constructs. In the simplest, most portable implementation, a protothread is a function, the body of which is nested into a switch statement. Depending on the value of a thread state variable (analogous to a program counter), a call to the function implementing a protothread will transfer control to different case labels at the code nested in the switch block. This way, a protothread can block or yield by advancing the program counter followed by a return. Next time the function implementing the protothread is called, thread execution will resume at the next case label.

The immediate advantage of using protothreads is that explicit state machines can be eliminated from event-driven code. The code is typically shorter (3 times, according to [20]), and easier to understand due to its linear control flow. Protothreads are portable across compilers and across platforms: no special libraries, no OS support, no assembly coding is required, only a standard-compliant C compiler. The execution overhead of a protothread is in the order of a few processor cycles: a jump, the destination address of which is dependent on a variable. Since the effective execution model is still event-driven, such system can be implemented using a single stack. In fact, the memory required by a protothread is merely a byte or word that stores the program counter.

On the downside, protothreads have a number of shortcomings. First, local automatic variables in a protothread are not retained between subsequent calls to the thread, because the stack is unwound every time the protothread blocks. The easiest way to overcome this limitation is declaring variables local to a thread as static, allocating them in the data section of the memory instead of the stack frame. In case of protothreads that need to be reentrant, however, further workarounds may be required.

Second, Standard C does not allow for nested case statements. This limitation disallows using switch statements within protothreads, because the body of a protothread gets nested in a preprocessor-generated switch statement.

Third, compiler related problems may also arise. Although protothread constructs are valid C programs, some compilers may or may not honor arbitrary C code within a body of a switch statement, along with arbitrarily placed labels. Furthermore, compiler optimizations might also be source of problems. For example, the compiler might decide to move a variable into a register, which results in the value of such variables being lost between subsequent calls to the thread. Similarly, the compiler might decide to move loop invariant code out of the loop, which could break a protothread that blocks in that loop.

NutOS

NutOS is the operating system of the EtherNut device, a tiny, Ethernet-enabled device designed for networked embedded applications. NutOS supports cooperative multithreading, and provides kernel-user separation. In contrast to Contiki, threads in NutOS *are* managed by the operating system: when a thread reaches a yield point, it is the OS scheduler that decides which thread to schedule next. There is a per thread stack and a thread control block belonging to each thread, which, obviously, increases the application's overall memory requirements. The EtherNut, based on the same Atmel microcontroller as most of the state-of-the-art wireless sensor nodes, is, however, equipped with an external SRAM chip (32kB for the first version, 512kB for subsequent versions), which relieves programmers and OS designers from dealing with memory constraints.

Mantis OS

A good example of operating system concepts scaled down to a wireless sensor platform is the MANTIS OS (MOS) [1, 8]. The primary design objective of MOS was to provide productivity: ease of use and a moderate learning curve. As the programming environment that MOS provides is similar to that of general purpose OS'es, little training is required for programmers to achieve productivity and be able to expedite prototyping of wireless sensor network applications.

The MANTIS OS has a layered architecture. The kernel consists of a preemptive scheduler, device drivers, network stack, etc., while the applications are running in user space. Since the target platforms have no MMUs and do not offer privilege separation, MOS does not provide memory protection or virtual memory, unlike general purpose operating systems.

MOS provides preemptive multithreading with time-slicing. Multithreading is a very expressive abstraction for programming concurrent applications, because threaded code has linear control flow. This, however, comes at a cost. First, a separate stack is required for each thread, which results in significant memory usage. Considering that a typical target platform has only 4kB of RAM, this imposes a tight limitation on the number of concurrent threads the OS can support. Second, data that can be modified by concurrent threads must be protected with locks to assure mutual exclusion. Locking, on one hand, decreases system performance, on the other hand, it is a very hard task and a common source of bugs.

RETOS

RETOS [13] is built along traditional OS concepts and provides kernel and userspace separation, emulates memory protection on MMU-less hardware and supports multithreading. To achieve this in presence of resource constraints, RETOS relies on a static analysis toolchain.

As multithreading inevitably dictates that a separate stack is required for each thread, allocating a fixed-size stacks would be either wasteful or unsafe. With static stack depth analysis techniques, however, it is possible to estimate stack sizes for each thread, and allocate memory for the stacks accordingly. RETOS employs a stack depth analyzer that runs on binary code. To further decrease the memory requirements, RETOS maintains only one kernel stack. This implies that threads executing in kernel mode cannot be preempted.

On MMU-less microcontrollers, there is no hardware support to protect kernel data structures from accidental corruption by malignant applications. For this reason, RETOS employs software based memory protection mechanism, which is a combination of complementary static and dynamic safety checks. In the compiled binary, the destination fields of memory write machine instructions are inspected. Similarly, the source fields of read instructions can also be checked to prevent data access by untrusted code. While it is possible to statically check pc-relative jumps, direct and indirect addressing, safety of indirect addressing can only be assured runtime.

Virtual machines

While multithreading virtualizes the microcontroller by presenting an execution context to the threads in which they can operate with the assumption of exclusive usage of the MCU, virtual machines (VM) provide a similar, but higher level of abstraction. A VM provides a virtual CPU (or CPUs) with an instruction set which is different from that of the host platform.

Several virtual machines have been proposed for wireless sensor nodes. VM* [47] and CVM [21] are derivatives of the Java virtual machine, targeting resource-constrained devices. Maté [54] is a stack based VM for sensor nodes that aims to provide high code density and thus inexpensive code updates. As an extension of Maté, application specific virtual machines (ASVM) [55] have been proposed to allow programmers to introduce domain specific instructions in the virtual machine's instruction set.

Although programs running on top of VM execute three to ten times slower than native code, the flexibility these tools provide through portability, development time, and code update costs, often prove to be a reasonable tradeoff in WSN applications.

2.1.3 State machines

Object State Model

The Object State Model (OSM) [46] employs attributed state machines to express event-based program behavior. The application of finite state machine (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 extensively borrows concepts from previous work on state machine based programming formalisms, for example, hierarchical modeling, parallel composition and broadcast-based communications from Harel's StateCharts [42], concurrent events from SyncCharts [3], as well as explicit state variables from Finite State Machines with Datapath (FSMD) [29].

Through support for hierarchy and attribution of states with shared variables, OSM offers efficient allocation of shared state, leveraging the knowledge of the lifetime of variables. This way, OSM eliminates the need for manual stack management.

Semantically, an OSM specification can be mapped to the synchronous reactive (SR) [25] model of computation. In practice, OSM specification is translated to Esterel [7], a synchronous language[6], which then can be compiled into efficient C code by the Esterel compiler.

The SR paradigm offers deterministic concurrency through static scheduling, which computed by the compiler. In the SR model of computation, programs are executed in a lockstep manner with one or more clocks. In particular, Esterel modules communicate with signals, the presence of which is checked by the module in the beginning of every step. Within a step, all modules that check a specific signal will see it either present or absent, but never both. Signals do not persist across steps: a signal is present in a reaction if and only if it is emitted by the program or is made present by the environment.

One apparent shortcoming of the SR paradigm in connection with wireless sensor nodes is its high latency in serving external events. From the implementation point of view, interrupts need to be buffered between clock ticks. This implies that if an interrupt occurred shortly after the synchronous clock fired, the corresponding signal will only be present at the next clock tick. This delay can be conservatively estimated by adding the worst case execution times of each module. There are, consequently, certain time-critical operations that are common to wireless sensor nodes, e.g. radio communications or event timestamping, which are cumbersome to implemented using the OSM toolchain.

2.1.4 Macroprogramming

All previously mentioned programming models assume that sensor network programming is done in a bottom-up manner. Namely, programs are written from the point of view of individual node in the network, which cooperates with others to solve the given problem. There is, however, a number of systems that use a top- down approach to sensor network programming. In contrast with the bottom-up approach, where the programmer writes the behavior of each node to achieve a global behavior, top-down approach is to write global behavior and node-level behavior is automatically generated. Programs are specified in a high level language which assumes that the sensor network is the target platform. That is, programmers implement a central program that, conceptually, has access to the entire network. This top-down approach allows programmers to focus on high- level algorithms, hiding the low-level details (tasking individual sensors, communications, resource management, etc.). It is typically a compiler or a runtime (or a combination of the two) which maps the high-level specification to efficient code running on individual nodes. In the sensor network literature, this style of programming is termed macroprogramming.

Sensor network as a parallel/distributed computer

Regiment [62] is a functional programming language that allows for centrally programming wireless sensor networks. It is built around a high-level programming concept called *abstract regions* [78], an abstraction that encompasses neighborhood discovery, enumeration and data sharing. Regiment models sensor data generated within a user-specified region as a data stream. Being a functional language, Regiment has a number of advantages over declarative languages. Regiment is side-effect free. Consequently, programs written in Regiment will be free from software errors resulting from erroneous variable updates. Since it is not possible to modify the value of a variable once it is assigned, the concept of global state does not exist in Regiment. This relieves the compiler from explicitly managing global state, and yields the way to various compiler optimizations (tasking of nodes based on topology, etc.).

The Pleiades [48] programming language has similar objectives, however, it takes an imperative approach. Pleiades extends the nesC language [32] with the `cfor` construct, that specifies parallel execution of the body of the `cfor` loop across multiple nodes. Parallel code has to be serializable, that is, the global state after the loop has completed must not depend on the interleaving of execution within the `cfor` block. The most important contribution of Pleiades is that it features a centralized programming model and pushes the burden of concurrency control and synchronization to the compiler and runtime.

Database centric approach

Cougar [80] and TinyDB [59] treat the sensor network as a distributed database. Database queries can be expressed in an SQL-like, declarative language. Queries may include sensor attributes (e.g. individual sensor readings), arithmetic functions of attributes, selection with attributes as arguments, as well as aggregate functions of attributes (average, sum, minimum or maximum). Both Cougar and TinyDB provide support for temporal and data streaming concepts that specify when sensors should be sampled, as well as the frequency and the duration of sampling. In addition, TinyDB supports event based queries, which are triggered or terminated by other queries or by software running on the sensor node.

Although, SQL queries are interpreted at the base station (typically a PC) and mapped to low-level commands which are disseminated in the sensor network, the evaluation of conditionals and the computation of aggregates are pushed into the sensor network.

It is important to note, however, that these systems were designed for simple monitoring applications. TinyDB and Cougar lack support for arbitrary computation at the nodes; and obviously, communication primitives are not exposed through the query interface. Therefore, database centric approaches are not suitable for general-purpose application development.

2.2 Motivation

While many operating systems approaches have been shown to be feasible for wireless sensor nodes, the WSN research community is somewhat biased towards the event-driven paradigm. The primary reason for this is that an event-driven runtime is easy to implement and to port, and that event-driven systems typically have a small memory footprint. On platforms with very limited physical memory, other approaches (e.g. multithreading) may prove prohibitively expensive in terms of resource usage, leaving an event-driven operating system the only feasible choice.

Programming event-driven systems, however, can be very difficult. Since event-driven programs need to be implemented as a set of event-handlers, the logical sequentiality of event invocations is not possible to express in traditional programming languages, such as C. Hence, programs often have to be implemented as explicit state machines, which is a tedious and error prone task (often referred to as *manual control flow management*). Furthermore, as there is no language support for information sharing between event handlers, programmers have to manually allocate the shared variables, emulating the C stack. Both manual control flow and stack management can grow very complicated as the size and complexity of the application increases. Hence, implementing event-driven applications that are both reliable and memory efficient requires a major effort.

I demonstrate the inherent complexity of event-oriented programming through an example, which shows that managing control flow manually can be challenging, even in simple applications.

2.2.1 Motivating 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 (Fig. 1) describes the procedure that writes a variable-length packet to the bus, using the byte-oriented hardware interface:

```
procedure{I2CPacket.writePacket}{length, data}
    call I2C.sendStart
    wait for I2C.sendStartDone
    for index = 0 to length
        call I2C.write(data[index])
        wait for I2C.writeDone
        index = index + 1
    endfor
    call I2C.sendEnd
    wait for I2C.sendEndDone
    signal writePacketDone
end
```

Figure 1: Pseudocode of a packet-oriented I2C driver. This code illustrates the packet writing functionality only. The control logic of reading a packet is similar.

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).

As a result, a corresponding event-driven solution is typically much more involved than the above pseudocode: the I2CPacket module of TinyOS 1.1 (with packet read and write functionality), for example, consists of more than a hundred lines of code.

2.2.2 Problem formulation

The purpose of this work is to create a tool that

- allows for event-driven programming by describing conceptually concurrent threads of computation in an intuitive way,
- automates tedious tasks such as manual control flow and stack management,
- maps it to efficient code, that does not rely on run-time multithreading support and avoids the need for stacks for each thread, and
- protects from common programming errors.

2.3 Organization

This chapter is structured as follows. First, in Section 2.4 I present TinyVT's approach to providing a thread abstraction on top of an event-driven runtime. I define and characterize TinyVT threads, and state the assumptions on the event-driven runtime hosting them. The TinyVT language, an extension of C, is presented in Section 2.5. Then, Section 2.6 gives a detailed description on the TinyVT compiler, outlining how the thread abstraction is resolved to simple C code. To illustrate TinyVT's capabilities, Section 2.7 presents a case study, implementing a data collection application using TinyVT threads. The chapter concludes with discussing TinyVT's strengths and limitations with respect to multithreading and event-oriented programming, respectively.

2.4 Approach

My approach to eliminating manual control flow management and manual stack management from event-driven programming is the following.

Related event handlers, for instance, those that constitute the implementation of the same service, are wrapped in a container and handled as a unit. To every container, local state is assigned, which evolves in reaction to incoming events. The container supports dispatching event handlers not only based on event kind but also on local state. In order to achieve this, multiple event handler implementations can be defined for the same event type within a container, which are associated to different configurations of the local state. In response to an event, the container carries out a computation, which computes a return value and changes the local state (and thereby the behavior of the thread in response to the next input event).

The definition of this evolving state and evolving behavior (i.e. the control flow) is given using well-known, well-understood programming constructs common to procedural languages, such as C. Specifically, I extend the C language with and additional construct to define such a container, called the *thread definition*, which, like the implementation of a function, has a linear control flow, and offers nested scopes that may contain local variables with automatic storage duration, alleviating the need for manual stack management. The thread definition provides a programming abstraction which emulates that the thread definition has an independent local thread of execution, thereby relieving the burden of manual control flow management from the programmer. Within the thread definition, the language extension provides a special `await` statement, which is an opaque, blocking statement from the perspective of the local execution thread. The await construct wraps an exit point of an event handler and an entry point of another handler (and part of the event handler's code) in one statement. This is similar to how a function invocation expression shields that control is passed to the implementation of the function and later returned to the caller – all this during the execution of the function invocation expression. From the thread's perspective, the await statement is like any other statement and is executed in line with other statements, as the thread's control flow defines it.

This programming abstraction, called TinyVT, is achieved with a language extension and the corresponding compiler, which translates TinyVT specific language constructs to plain C code, which can operate on top of a lightweight event-driven execution engine.

2.4.1 TinyVT's thread abstraction

A *TinyVT thread* is a computational agent with local state, conceptually independent thread of execution, and source code with linear control flow. From the thread's point of view, it is perceived that the TinyVT thread has its own thread of execution, executing the statements that constitute the thread's source code sequentially, or with jumps between them if the C control structures in the source code define so.

In contrast with traditional multithreading, where the operating system or a user-space threading library creates an abstraction of a virtual processor on which the thread is exclusively executing, the abstraction of an independent execution thread in TinyVT is provided by the TinyVT language. Threads are *"compiled away"* (resolved) by the compiler, reducing the thread to a set of event handlers. Therefore, TinyVT is a *compiler assisted threading abstraction*.

Control flow of a thread is defined over statements in the source code of the thread, not over binary machine instructions. The key enabler of TinyVT's thread abstraction is the `await` statement, which is an opaque, blocking statement in terms of the thread's local control flow, but a wrapper of an exit *and* an entry point from a lower-level, fine-grained perspective. The conceptual, local control flow of the thread is unaware of the fact that, while an `await` statement is being executed, control is first passed to the environment and then it returns to the thread, all within the same opaque statement.

In TinyVT terminology, *yielding* is returning control to the caller to the originator of the event that triggered the current execution step of the thread. It is important to note that TinyVT threads have explicit yield points. All yield points are associated with await or yield statements in the thread's source code. Since the environment is event-driven where event handlers run to completion, calls to external functions never cause the thread to yield. Therefore, the programmer has complete control over yielding. This is unlike in traditional multithreading, where library functions may block and yield, without the caller being aware of it or being able to control it.

Program vs. execution of program

In TinyVT terminology, thread, without ambiguity, can refer to the source code of the program *and* to its conceptually independent thread of execution, because there is a direct correspondence between the two. TinyVT threads are static in the sense that they are implicitly instantiated and cannot be spawned dynamically at runtime. Therefore, the source code of a TinyVT thread can only have one thread of execution, and the thread of execution is, conceptually, local to the thread. This is unlike the terminology of traditional multithreading, where distinction must be made between the program (a series of machine instruction), and the execution of the program, because the same code might be concurrently executed by multiple threads of execution, each having its own context (state).

Context

TinyVT threads also have local state, that is, context. The local thread state contains control flags and variables, including a variable storing the last yield point, which is much like an instruction pointer that specifies where the thread's execution should continue in response to an external event. Compiler-managed local automatic variables are also part of the thread's context, emulating the semantics of ANSI C's automatic storage duration by allocating them to static memory. It is important to note that, in TinyVT, threads have no stacks associated with them, unlike in traditional multithreading. A TinyVT thread will always use the stack of the triggering event to store local variables, pass parameters and return values. Therefore, the context of a TinyVT thread does not contain a local, dedicated stack.

Interaction with the environment

From the environment's point of view, execution of a thread progresses in uninterrupted execution steps, in response to function calls (events) from the environment. The thread maps each input event to

- an execution of a series of statements in the thread's source code, which varies depending on the local state,
- a new local state, and therefore, a new behavior which will govern the response to the next event,

- and a return value, returned to the originator of the input event (i.e. caller of the event handler function).

Therefore, a TinyVT thread constitutes a higher level of abstraction than just a set of event handlers. Since a thread has local state, the thread has a history: the behavior of the thread in response to an external event depends not only on the kind of the event received, but also on previously received events.

The thread's environment perceives the TinyVT thread as a set of event handlers (the "program"), while the abstraction of a local, independent thread of execution (the "execution of the program") is not visible from the outside. From the environment's point of view, the thread is a passive software artifact, the execution of which is driven by the environment. That is, the TinyVT thread (i.e. event handlers corresponding to the thread) has the control only when it is executing in response to an event from the environment (i.e. a function call to one of the event handlers), and relinquishes control when the execution of the event handler completes.

A TinyVT thread perceives its environment as a set of functions, which the thread may invoke. If multiple TinyVT threads are defined in a program, they perceive each other as part of the environment, that is, as a set of event handlers.

2.4.2 Assumptions

A TinyVT thread thread is running on top of a lightweight event-driven runtime. The thread has the following assumptions on the runtime environment.

- **Initialization service.** The thread cannot accept events before it is initialized. The runtime environment must guarantee to call the thread's initialization function before sending events to the thread (i.e. calling functions that are implemented as event handlers inlined in the await statements of the thread). The name of the initialization function is identical to the name of the thread, it has void return return type and an empty argument list.

- **Event dispatcher.** The execution of a TinyVT thread is driven by its environment. The thread has the control only if the environment passes it to the thread by sending an event. From the environment's point of view, the thread is a passive software entity: it executes only in response to external events. The thread assumes all calls to the event handlers it implements originate, directly or indirectly, from the event-driven dispatcher, which is assumed to be part of the thread's environment. The dispatcher dispatches events in a serialized manner. The next event is not dispatched until the currently executing handler returns. Asynchronous execution contexts, for example, interrupt handlers, are not allowed to call into the threads. They may interact with the dispatcher, placing events to the dispatcher's event queue, which will be dispatched, in a deferred manner, to the TinyVT thread that implements a corresponding handler.

- **Deferred procedure call service.** To implement the TinyVT specific yield and ireturn statements (see later), the environment is required to provide a deferred procedure call (DPC) service. The DPC service must provide an API to request the deferred execution of a function, the handler of which is implemented by the thread. The DPC service must guarantee that DPC request is serviced after the function call requesting the DPC returns to the requester. This will imply that the deferred event will be sent to the thread after the event handler that requested the DPC completes. In most event-driven systems, the event dispatcher provides a DPC service, to allow for software generated events.

- **Runtime error handler.** A non-reentrance violation, or reception of an event on which the thread does not explicitly block, but otherwise reacts to, causes a runtime error. The environment assumed to provide a means to handle run-time errors, by providing a function a call of which causes thread execution to halt. The C code generated from a thread will invoke this halt function if a runtime error occurs.

2.5 Language constructs

TinyVT extends the C language with five additional keywords: `thread`, `yield`, `await`, `dreturn` and `ireturn`. They are used to define a TinyVT thread, explicitly specify yield points, perform blocking wait, and set the return values of the events that drive the thread's execution. This section defines the syntax of these language constructs by extending the phrase structure grammar of the ANSI C language (as specified in Appendix A of the C99 standard [28], pages 409–416). Explanations of TinyVT's language constructs are presented with the corresponding grammar segments. Words in italics are nonterminals and non-literal terminals, typewriter words and symbols are literal terminals. The opt subscript indicates that the nonterminal it follows is optional. The production rules of the C grammar are not repeated in this section, only those C production rules are described that are changed in TinyVT.

2.5.1 Thread definition

The thread definition is used to define a piece of code with an independent, linear control flow. The thread definition has a name and an implementation:

thread-definition:

'`thread`' *identifier compound-statement*

Thread definition starts with the `thread` keyword. Since `thread` is a keyword, its use as an identifier is not allowed in the translation unit.

The thread name is used by the TinyVT compiler to mangle the identifiers in the generated code (to include the thread name in the identifiers). The purpose of the mangling is to allow multiple TinyVT threads coexist in one translation unit, thereby guaranteeing that the identifiers used internally in the generated code are unique to each thread.

A thread definition implicitly adds a function definition to the global scope, with an empty argument list and void return type. The name of the function is identical to the name of the thread. Therefore, a translation unit must not have two thread definitions with the same name.

This function is called by the execution environment to initialize the thread, that is, bootstrap the thread's execution by running it until the first await statement. If multiple threads are defined in the same translation unit, their initialization order depends on the semantics of the execution environment.

The compound statement holds the implementation code of the TinyVT thread, and defines a scope under the global scope. Its syntax is identical to that of C compound statements. Additional TinyVT specific statements (`await`, `yield`) can also be used within the implementation of the thread and its nested compound statements (except inside inlined event handlers, see later), while the use of return statement is not allowed. Outside thread definitions, await and yield are not allowed.

TinyVT extends the syntax of C statement as follows:

statement:
> *await-statement*
> *yield-statement*
> *dreturn-statement*
> *ireturn-statement*
> *labeled-statement*
> *compound-statement*
> *expression-statement*
> *selection-statement*
> *iteration-statement*
> *jump-statement*

The defined thread is static, that is, it is implicitly instantiated. After the program is loaded and initialized, the thread is blocked at the first await statement and ready to accept events from its environment.

Non-static variables declared within the thread's implementation have automatic storage duration, similarly to local non-static variables in function definitions. Therefore, they are (conceptually) deallocated after the thread's execution leaves the scope in which the variable is defined. The semantics of static variables within threads is identical to that of static variables in C.

Thread definitions cannot be nested in C language constructs: they must be defined top level in the translation unit. Therefore, TinyVT alters the C syntax of translation unit by extending the *external-declaration* rule as follows:

translation-unit:
> *external-declaration*
> *translation-unit external-declaration*

external-declaration:
> *thread-definition*
> *function-definition*
> *declaration*

A thread definition implicitly adds a function definition for every event kind to the translation unit's global scope, for which an event handler exists inlined in any of the thread's await statements.

2.5.2 The yield statement

TinyVT provides the yield statement to allow the temporarily suspension of the thread's execution to allow other software artifacts (external to the thread), managed by the thread's event-driven scheduler, to execute. From the thread's point of view, relinquising control and then the return of control back to the thread is hidden. The yield command is opaque: conceptually, yield is a blocking statement, that is, execution of yield completes only after control is passed back to the thread.

The syntactic production rule of the yield statement is the following:

yield-statement:
```
                    'yield'
```

Since yield is a keyword, its use as an identifier is not allowed in the translation unit.

2.5.3 The await statement

The await statement allows temporarily relinquishing control and blocking on an external event, on which thread execution resumes. The await statement has the following syntax:

await-statement:

'`await`' '`(`' *function-definition-list*$_{opt}$ '`)`'

Where the production rule for *function-definition-list* is:

function-defintion-list:

function-definition
function-definition-list

The use of `await` as an identifier is not allowed throughout the translation unit: it is a reserved keyword. The function definitions inlined in the await statement (also referred to as the *inlined event handlers*) specify

a. the kinds of events on which the the await statement should block, and

b. the code that should be executed in response to the event that resumes thread execution.

The name and the type signature of the inlined function definition specifies the event kind, while the compound statement defines the the code with which thread execution resumes when an event of the specified kind is received while blocking at the enclosing await statement.

Multiple handlers of the same event kind may exist within a thread, but only in different await statements. One await statement must not have two inlined function definitions with the same name and signature. No inlined await handlers with the same name but different signature are allowed within a thread: if multiple event handlers with the same name are present (inlined in different await statements), they must have identical signatures.

The set of all event kinds specified in inlined event handlers within a thread define the list of input events to the thread, i.e. the input events to which the thread reacts, either by resuming execution or with a runtime error. (A runtime error occurs on reentrance violation, or when the event is not handled at the current await statement.)

For every event kind to which the thread reacts, a function definition, with the corresponding name and type signature, is created in the global scope. This has several consequences:

- No two threads, within the same translation unit may react to the the same event kind.
- Function definitions with the same name must not exist in the global scope.
- If a function declaration (forward declaration) with the same name exists in the global scope, the signature of the function definition that corresponds to the event kind must be identical to the signature of the global function declaration.

From within the thread, invoking an event handler defined by the thread will always result in runtime error (reentrance violation). However, the thread may take the address of the event handler and use it, i.e. passing it as a function argument.

It is possible that the await statement contains no inlined event handlers. If no inlined event handlers are given, the thread's execution blocks permanently at the await statement, that is, the thread will never resume and the execution of the await statement will never complete.

The C return statement cannot be used within inlined event handlers, only the TinyVT specific dreturn and ireturn statements are allowed. The use of one or the other is mandatory: every exit point of the function's body must be explicitly designated with ireturn or dreturn.

TinyVT does not allow for non-local jumps and targets of non-local jumps within inlined await handlers, since this would violate the requirement that every exit points within the inlined event handler must be explicitly marked with dreturn or ireturn. That is, goto statements with target labels outside the event handler, continue and break statements that correspond to iteration statements outside the await statement are not permitted. Also, a goto or a switch statement that is outside the await statement cannot reference a label inside an inlined event handler.

2.5.4 Immediate and deferred return

Conceptually, a TinyVT thread never returns. For this reason, the return statement is not allowed within the thread's implementation. However, the external events that drive the thread's execution, do return – although such a transfer of control flow outside the thread is always encapsulated in a TinyVT statement (`await` or `yield`), which also contain entry points through which control comes back to the thread. Event handlers inlined in await statements must use either dreturn or ireturn to specify the handler's return value. All exit points of an inlined event handler must be marked with either dreturn or ireturn, even if the function returns void. Their syntax is the following:

dreturn-statement:
 'dreturn' *expression*
 'dreturn'
ireturn-statement:
 'ireturn' *expression*
 'ireturn'

Both ireturn and dreturn are keywords; using them as identifiers is not allowed. They can only be used within inlined event handlers within an `await` statement. An ireturn or dreturn may only have an expression if the return type of the enclosing event handler is non-void.

When a dreturn statement (which stands for deferred return) is executed, the enclosing event handler's return value is set to the value of the expression (if given), however, control is not returned to the caller of the event immediately. Execution of the thread continues with the statements following the enclosing await statement until the next yield point is reached. Control is then returned to the originator of the event that triggered the execution step.

In contrast, ireturn means immediate return. If an expression is given, it is evaluated and the return value of the event handler is set to the value of the expression, and control is returned to the originator of the event (i.e. the caller of the event handler). The scheduler will resume the execution of the thread starting with the first statement that follows the await statement.

2.6 The TinyVT compiler

TinyVT provides language features to express a threading abstraction. Conceptually, a TinyVT thread has its own independent thread of execution, which executes the statements of the thread according to the thread's local control flow, as defined by the order of statements and the semantics of C control structures (`if`, `for`, `while`, etc.).

The execution of a TinyVT thread is driven by interaction with the thread's environment. In response to an external stimulus, execution of a blocked thread resumes at the last yield point. The thread runs until control reaches a yield point, it blocks and returnins control to the originator of the triggering stimulus. Yield points are hidden within statements, shielding the thread from the fact that control leaves from, and later, returns to the thread. The key enabler of the threading abstraction is the `await` statement, which is, conceptually, a blocking statement, while, from the environment's point of view, it is in fact a yield point and an entry point at the same time.

Thread execution progresses in uninterrupted steps, executing a series of instructions from one yield point to another. Between the execution of two consecutive steps, the thread is blocked. An execution step is always triggered by an external event, which manifests itself as a function call from the environment.

The TinyVT compiler is a source-to-source translator. By analyzing the source code of a TinyVT thread, it generates blocks of C code that corresponds to execution steps, and synthesizes the logic which sequences the execution of such blocks of code, dispatching them based on input event kind and thread state. This way, the transformation decreases the level of abstraction of the program code, by resolving virtual threads written using TinyVT to a set of event handlers given in the C language.

The TinyVT compiler performs the the following tasks:

- **Resolving syntactic shorthand notations.** TinyVT threads can be rewritten without using the `yield`, `dreturn` and `ireturn` statements, such that the only TinyVT specific language construct the resulting code contains is the `await` statement. Subsequent compiler tasks will be simpler to implement after this syntactic normalization.

- **Allocation of shared variables.** TinyVT allows for nested scopes with automatic local variables. Since, however, the stack is unrolled every time the thread yields, automatic variables declared in a compound statement that contains a yield point cannot be allocated on the stack. To emulate the semantics of ANSI C's automatic storage duration, the TinyVT compiler moves the declarations of such variables to the global scope. Based on the observation that variables of non-overlapping scopes are never alive at the same time, they can be allocated to the same memory region. The TinyVT compiler achieves this by generating a type of nested `struct`s and `unions` that encapsulates all such automatic declarations, and allocates a variable of this type in the global scope. References to the automatic variables are replaced by references to the corresponding members of the generated data structure.

- **Construction of local control flow graph.** To facilitate program analysis and code generation, a control flow graph is generated which captures local control flow within a thread, but hides the fact that control flow may temporarily leave the thread, by enclosing the points where the control flow leaves the thread and the points where it returns to the thread in `await` statements. The local control flow graph (LCFG) is built such that every `await` statement of a thread becomes a node of the graph, allowing for easy querying of yield points and input event kinds. Each non-await node of the graph contains a sequence of syntactically correct C statements, and an optional conditional expression to support branching. Since yield points may appear nested within C control structures such as loops (`for`, `while`, etc.) or selection statements (e.g. if statements), the block of code between yield points (i.e. between `await` statements) might not be a syntactically correct series of C statements. For example, it may contain `do` but no the ending `while`. The compiler, therefore, preforms code transformation on the TinyVT source to generate multiple LCFG nodes from such blocks of code by rewriting the fractured C control structures to syntactically correct C statements. Finally, a pruning pass ensures that no unreachable blocks exist in the LCFG to avoid unnecessary work in further compiler passes.

- **Identification of yield points.** TinyVT requires that yield points be explicitly specified in TinyVT threads, and provides language constructs such as `await`, `yield` and `ireturn` to achieve this. The latter two are rewritten using `await` in the syntactic normalization pass, and every `await` statement is converted to a node in the local control flow graph. Identifying yield points, thus, falls back to locating await nodes in the LCFG.

- **Enumeration of input events.** When an input event occurs, execution of the thread resumes starting with the first instruction of the corresponding event handler inlined in the `await` statement at which the is thread blocking. Await statements may contain more than one event handlers. When an event occurs, control will be passed to the handler the name and signature of which matches those of the input event. Therefore, to enumerate all possible input events of a thread, the compiler must inspect all event handlers in all `await` statements. This is achieved by visiting all await nodes of the LCFG and building the set of function signatures of inlined event handlers encountered within the `await` statements.

- **Generation of C code implementing primitive blocks.** When an event occurs, the execution of the thread progresses uninterrupted from one yield point to the next one. The corresponding C code is generated by dumping the C code inside non-await nodes of the call flow graph, which will be dispatched from the generated event handlers (see later).

- **Generation of code that maintains thread state.** TinyVT threads are not always input enabled. First, TinyVT threads are not reentrant, that is, no inputs are accepted while the thread is executing. Inputs are only allowed after the execution has reached a yield point and the thread is blocked. Second, even when the thread is blocked, it accepts only those events that are explicitly specified in the `await` statement corresponding to the current yield point. The compiler has to keep track of thread state in order to protect against the violation of the above input rules. To protect against reentrance violations, it generates code that sets a flag when the thread resumes and clears the flag when the thread yields. When an event occurs that finds the flag set, a runtime error is raised. Also, the position of the last yield point has to be stored as part of the thread state, such that when the thread resumes, the control can be passed to the inlined event handler that matches the input event. If the event is not accepted at the current yield point, a runtime error occurs.

- **Synthesis of local control logic.** The compiler synthesizes the logic that, on an external event, inspects the thread state and passes control to the event handler with the same event kind, that is inlined in the `await` statement at which the thread last yielded. Before the next yielding, the generated code writes the position of the new yield point to the thread state and returns control to the caller of the triggering event.

2.6.1 Pattern based code transformation

To simplify further code analysis and translation, the compiler applies a series of pattern based code transformation steps after parsing the TinyVT source.

First of all, the compiler attaches an `await` statement with an empty body (i.e. with no event handlers inlined) after the last statement in the source code of the thread. This assures that if control reaches the last statement of the thread, the thread blocks permanently. If this *implied* trailing `await` statement later turns out to be unreachable, no corresponding code will be emitted by the compiler.

Then, the compiler removes syntactic sugar: statements, such as `yield` and `ireturn`, which are shorthand notations for alternative, more verbose, but semantically equivalent constructs. Then, the compiler carries out a pattern-based code transformation partly resolving `dreturn` statements to C code.

Although, for efficiency, these pattern based transformations are applied to the abstract syntax tree of the TinyVT thread, they are simple enough to be implemented with text processing tools, using, for instance, regular expressions. Therefore, in order to increase the readability of the description of the transformation passes, the illustrative examples will be given in a textual form.

Identifiers in declarators that are generated by the TinyVT compiler and that are used internally within the generated code are prefixed with double underscore, and are mangled to include the name of the thread, as well. This mangling is required to prevent such declarations, generated from multiple threads within the same translation unit, from colliding with each other in the global namespace. This mangling, however, is omitted from the source code examples throughout this section, in order to help comprehension.

Resolving yield and ireturn statements

The first pass of the TinyVT compiler resolves syntactic shorthand notations. This pass rewrites all `ireturn` and `yield` statements, leaving only two TinyVT specific statements in the thread code: `await` and `dreturn`. Resolving `yield` and `ireturn` statements requires only pattern based source translation, where the structure of the program remains the same, and the input and output of the transformation are semantically equivalent.

Resolving ireturn statements

According to the semantics of the TinyVT language, the `ireturn` statement returns control to the caller of the enclosing event handler without executing any statements that follow the enclosing `await` statement. After yielding, the thread will be resumed by a deferred function call from the environment, starting the computation step with the first statement after the `await` statement that contains the `ireturn`. Hence, the `ireturn` statement is semantically equivalent to a `dreturn`, plus a `yield` statement immediately following the enclosing `await` statement. If `dreturn` and `ireturn` statements are mixed within an `await` statement, we must remember the type of return statement through which the inlined event handler was exited. The `yield` statement only needs to be executed if the event handler exited through an `ireturn` statement.

To demonstrate the transformation that resolves and `ireturn` statement replacing it with `dreturn`, `yield` and pieces of C code, consider the example in Fig. 2. This code is transformed to a semantically equivalent form by changing `ireturn` to `dreturn`, and inserting a `yield` statement after the `await` block.

```
await( void myEvent() {
 /* block of arbitrary C code
    containing no return statement */
   ireturn;
});
```

```
await( void myEvent() {
 /* block of arbitrary C code
    containing no return statement */
   dreturn;
});
yield();
```

Figure 2: The code on the left, containing an `ireturn` statement, and its semantically equivalent counterpart, shown on the right.

If, however, both `ireturn` and `dreturn` statements are used within an `await` block, the `yield` inserted after the `await` block has to be executed conditionally, only if the event handler exited through an `ireturn`. To remember the kind of return statement used, the translated code uses a flag, which is cleared when the event handler starts executing, and set only if an `ireturn` statement is used.

In the example in Fig. 3, both `ireturn` and `dreturn` statements are enclosed within the same `await` statement. Notice that the flag selected by the `__YIELD_AFTER_AWAIT_MASK` mask is cleared in the first statement of the event handler, and it is set in the code corresponding to the `ireturn` statement. The `yield` statement is executed only if the flag is set.

```
await( void myEvent() {
 /* block 1 of arbitrary C code
    containing no return statement */
   ireturn;
 /* block 2 of arbitrary C code
    containing no return statement */
   dreturn;
});
```

```
await( void myEvent() {
  __clear_flag(__yield_after_await);
 /* block 1 of arbitrary C code
    containing no return statement */
   __set_flag(__YIELD_AFTER_AWAIT_MASK);
   dreturn;
 /* block 2 of arbitrary C code
    containing no return statement */
   dreturn;
});
if( __is_set_flag(__YIELD_AFTER_AWAIT_MASK))
  yield();
```

Figure 3: The code on the left shows an await statement containing both ireturn and dreturn statements. The code on the right shows its semantic equivalent without ireturn statements.

The TinyVT compiler generates code such that `__YIELD_AFTER_AWAIT_MASK` designates one bit of the thread state. However, it can be easily implemented also as a variable on the stack, since its value is only used within one execution step.

The function definitions of `__clear_flag`, `__set_flag` and `__is_set_flag`, as well as the value of the `__YIELD_AFTER_AWAIT_MASK` constant, are generated by the compiler, and are discussed later in this section.

Resolving `yield` statements

The `yield` statement is syntactic sugar. It is equivalent to calling an external function requesting a deferred event, followed by an `await` statement that blocks the thread on that event. Replacing the `yield` statement with a function call and an `await` statement will result in semantically equivalent code.

The code generated from `yield` statements is specific to the deferred procedure call (DPC) service the environment offers. The code, which the `yield` statement is translated to, is given as part of the configuration of the TinyVT compiler.

Below are two examples of how a `yield` statement is resolved. The first example assumes that a DPC service provides the `dpc_request` function, with a function pointer as a parameter. When thread execution reaches the `await` statement, the thread yields. The

DPC service guarantees that it will dispatch the callback function, to which the function pointer points, in a deferred manner, which will resume thread execution.

```
dpc_request(&deferred_proc)
await(
  void deferred_proc() { dreturn; }
);
```

Figure 4: Resolving the yield statement (C)

The second example uses the TinyOS [44] scheduler as the DPC service, and is written in the nesC [32] language, which is a superset of C. The `post` statement requests the deferred execution of the `deferred_proc()` task, the function declaration of which is prefixed with the `task` specifier.

```
post deferred_proc();
await(
  task void deferred_proc() { dreturn; }
);
```

Figure 5: Resolving the yield statement (nesC/TinyOS)

Handling return values

Since TinyVT threads never exit (conceptually, at any point in time, a thread is either executing or blocking on an event), the use of the C `return` statement is not allowed within a thread. However, it is often required that results of a computation be returned to the originator of an event that triggered the current execution step. This can be achieved using the `dreturn` and `ireturn` statements. Every event handler must have a TinyVT return statement (`dreturn` or `ireturn`), even those returning void, in order to explicitly specify the type of return the programmer intends to use. Neither `dreturn`, nor `ireturn` is an implied default.

Since `ireturn`s are reduced to `dreturn`s in the previous compiler pass, it is sufficient to discuss the resolving of `dreturn`s below.

The type of the returned data is specific to the triggering event: the return type is part of the function definition of the event handler which is inlined in the the `await` statement. To guarantee that the type of the returned data always matches the return type of the event handler, a `dreturn` (or an `ireturn`) statement is allowed only within event handlers, embedded in an `await` statement. TinyVT disallows `dreturn` (and `ireturn`) statements outside inlined event handlers, because the execution of such code may be triggered by multiple events, potentially requiring different return types.

It is important to note that the `dreturn` statement does not constitute a yield point. When control reaches a `dreturn` statement, control is not returned to the originator of the event that triggered the current execution step. Instead, the return value (if `dreturn` is followed by an expression) is stored in a temporary variable, and control is passed to the first statement that follows the enclosing `await` statement. It is the temporary variable that is returned when the execution reaches a yield point.

Generating the code that achieves this functionality is accomplished in two distinct steps. First, the `dreturn` statement is replaced by saving the return value (for non-void event handlers) to the `__rval` temporary variable (the declaration of which is generated by a subsequent code generation step), and by transferring control after the last statement of the event handler with a goto statement. The second step, which generates the code that declares and returns the temporary variable, is discussed later in the discussion of local control flow graph based transformations.

The following example in Fig. 6 illustrates the pattern based step of resolving `dreturn` statements.

```
await(
 char myEvent1() {
 /* block 1 of arbitrary C code
    containing no return statement */
    dreturn 'a';
 /* block 2 of arbitrary C code
    containing no return statement */
    dreturn toupper('b');
 }
 int myEvent2() {
 /* block 3 of arbitrary C code
    containing no return statement */
  dreturn 42;
 }
);
```

```
await(
 char myEvent1() {
 /* block 1 of arbitrary C code
    containing no return statement */
    __rval = 'a';
    goto __exit_myEvent1;
 /* block 2 of arbitrary C code
    containing no return statement */
    __rval = toupper('b');
    goto __exit_myEvent1;
__exit_myEvent1:
 }
 int myEvent2() {
 /* block 3 of arbitrary C code
    containing no return statement */
    __rval = 42;
    goto __exit_myEvent2;
__exit_myEvent2:
 }
);
```

Figure 6: Partial resolution of dreturn statements. The `await` statement on the left has two event handlers inlined, the first one returning a `char`, the second one and `int`. The result of the transformation is shown on the right.

Notice that, in contrast with the translation rules that resolve syntactic sugar, in this translation step the semantic equivalence of input and output source code is not retained, since this step accomplishes only a subtask of resolving `dreturn` statements. The resolution of `dreturn`s will be completed in a subsequent step.

Deterministic subexpression evaluation order

For many arithmetic expressions, as well as for function invocation expressions, the specification of the semantics of the C language does not prescribe a deterministic ordering of subexpression evaluation. As a consequence of this, the subexpressions of, for instance, the statement `i = f() + g();` can be evaluated in any order, varying from compiler to compiler, from platform to platform or even from run to run of the same binary. TinyVT, however, requires that the order of function calls made by the thread always be deterministic, therefore, expressions that include function call subexpressions with undefined evaluation order are not allowed in the generated code.

Currently, the compiler issues a warning message if such expressions are encountered, and leaves it to the programmer to modify the sources.

2.6.2 Allocation of automatic variables

The allocation of some variables within TinyVT threads, which have automatic storage duration semantics, cannot be handled by the stack-based automatic variable allocation scheme of the C compiler.

As the stack is unrolled every time the thread yields, automatic variables declared in a compound statement that contains a yield point cannot be allocated on the stack, because they would not be accessible in the code following the yield point. To tackle this issue, TinyVT provides a abstraction that allows for automatic local variables in threads with semantics identical to that of ANSI C's automatic storage duration irrespective of potential yield points in the enclosing compound statement.

The TinyVT compiler's automatic variable allocator algorithm consists of three stages. First, a scope tree is built, which reveals which automatic variables, if any, have to be allocated by the TinyVT compiler, and which of those are never active at the same time, and hence can be allocated to the same memory area. Then, the source code of a global hierarchical data structure, consisting of nested `struct`s and `union`s is generated, which encapsulates the declarations of automatic variables as nested members. Finally, declarations of compiler-managed automatic variables are removed from the AST, and all references to them are rewritten such that they refer to the corresponding members of the global hierarchical data structure, which is allocated in static memory.

Building the scope tree

In order to compute which variables may share the same memory region, the compiler computes the *scope tree* of the thread. Nodes of the scope tree represent either compound statements within the thread that contain at least one `await` statement, or variable declarations that are directly contained in such a compound statement. It is not necessary that the `await` statement be a direct child of the compound statement in the AST: any depth of nesting is sufficient. However, for every node in the scope tree that corresponds to a variable declaration, the compound statement that is the (direct) parent of the variable in the AST must be a node in the scope tree.

Nodes of the scope tree that correspond to compound statements are referred to as *scope nodes*, while those that represent variable declarations are called *declaration nodes*. Declaration nodes are always leaf nodes in the scope tree. A scope node is typically an internal node, except for the rare case when the corresponding compound statement does not contain any variable declarations.

The TinyVT compiler builds the scope tree from the abstract syntax tree (AST) which is the result of the pattern based transformation steps. Since the C language has constructs with nested statements (e.g. the statements specifying the branches of an if-else construct, or statements that constitute a compound statement between the '{' and '}' symbols), the AST is an inherently recursive data structure. Therefore, the parser implements the building of the scope tree with a set of recursive functions that visit the nodes of the AST recursively.

The prototype TinyVT compiler, including the scope tree builder, is written in Java. The scope tree building algorithm is explained in this section with excerpts (with simplifications) from the compiler source.

The scope tree is represented as a set of nodes of type `Node`. Each node object has a unique identifier (a positive integer). `Node` is an abstract class that has two subclasses: `ScopeNode` and `DeclarationNode`. Scope nodes contain a list of references to their direct children: the set of children of type `DeclrationNode`, or `ScopeNode`, can be accessed with the `getChildDeclarationNodes`, or `getChildScopeNodes` methods, respectively. Declaration nodes contain a reference to the AST of the variable declaration, which will allow for extracting the variable declaration as a string (`getDeclarationAsString` method), as well as the identifier of the variable declaration (`getIdentifier` method). The scope tree has a dedicated root node which is used as a handle to the scope tree of a thread.

The input of the scope tree building algorithm is the abstract syntax tree. To avoid lengthy description of abstract production rules that define the structure of the AST, the algorithms below use helper functions to extract information from the AST. The corresponding function names are self explanatory: the `AST getFirstDeclaration(AST ast)` function, for instance, returns the subtree corresponding to the first declaration of the compound statement the AST of which is given as a parameter. The scope tree building algorithm is bootstrapped by invoking the `visitCompoundStatement` function with the AST of the body of the thread definition as a parameter. It returns the root of the thread's scope tree.

First, the `visitCompoundStatement` method creates a new scope tree node for the compound statement. It iterates through all the declarations, generates the corresponding declaration nodes, and adds them to the scope node's list of children.

This is followed by an iteration over all statements that are directly contained within the compound statement. These statements can be compound statements, or other C control structures that contain nested compound statements.

The `getEnclosedCompoundStatements` helper function, called with an AST node, returns the list of such indirectly contained compound statements, but not the compound statements nested within them. If called with the AST of a compound statement as the parameter, the parameter is returned. Since the local automatic variables *can* be allocated on the stack if the parent compound statement contains no yield points, compound statements without nested `await`s do not have to be visited. The `hasAwait` helper function checks if the compound statement, the AST of which is given as a parameter, contains a nested `await` statement. For enclosed compound statements containing a nested `await` statement, `visitCompoundStatement` is called recursively. The returned node is added to the list of children of the current scope node.

```
Node visitCompoundStatement(AST compoundStatement) {
 ScopeNode scopeNode = new ScopeNode();

 AST currentDeclaration = getFirstDeclaration(compoundStatement);
 do {
   scopeNode.addChild(new DeclarationNode(currentDeclaration));
 } while((currentDeclaration =
            getNextDeclaration(currentDeclaration)) != null);

 AST currentStatement = getFirstStatement(compoundStatement);
 do {
  for(Iterator it = currentStatement.getEnclosedCompoundStatements();
            it.hasNext(); ) {
    AST enclosedCompoundStatement = (AST) i.next();
    if(hasAwait(enclosedCompoundStatement))
     scopeNode.addChild(
            visitCompoundStatement(enclosedCompoundStatement));
  }
 } while((currentStatement =
            getNextStatement(currentStatement)) != null)

 return scopeNode;
}
```

Figure 7: Building the scope tree.

Code generation

Variables in leaf nodes of the scope tree must be allocated to static memory. A straightforward way of doing this would be changing their declarations to static, or moving them to the global scope (with appropriate rewriting of names to avoid name collisions). However, it is easy to recognize that some of these local automatic variables are never active concurrently, because they are declared in non-overlapping scopes within the TinyVT thread. Allocating them to separate memory areas would be suboptimal, which is a critical issue, since potential target platforms are assumed to have only a few kilobytes of RAM.

The TinyVT compiler avoids this by allowing such variables to share static memory, by allocating them in a *global allocation structure*. This is achieved by wrapping declarations within a scope into a `struct`, and `struct`s corresponding to non-overlapping child scopes into an `union`. This, of course, can be done recursively, by allowing the `union`s representing the subscopes of a scope be members of the `struct` that corresponds to the parent scope. Please refer to Figure 8 for an example on how the generated allocation structure relates to the scopes containing compiler-managed automatic variables in the TinyVT source code.

```
{ /* scope1 */
  int a;

  { /* scope2 */
    char b;
    void* c;

    { /* scope3 */
      char d[10];
    } /* end scope3 */
    { /* scope4 */
      double e;
    } /* end scope4 */

  } /* end scope2 */
  { /* scope5 */
    int (*f)();
  } /* end scope5 */

} /* end scope1 */
```

```
struct {
  int a;
  union {
   struct {
    char b;
    void* c;
    union {
     struct {
      char d[10];
     } __scope3;
     struct {
      double e;
     } __scope4;
    } subscopes;
   } __scope2;
   struct {
    int (*f)();
   } __scope5;
  } subscope;
} __scope1;
```

Figure 8: The code on the left represents the conceptual scope structure of a TinyVT thread with compiler-managed automatic variables. The code on the right is the corresponding compiler-generated allocation structure. Notice the structural resemblance.

The allocation structure is generated by the following recursive function called with the root of a thread's scope tree as a parameter. For every scope node, a `struct` variable, with the name `__scopeX` is generated, where `X` is to be substituted with the unique identifier of the scope node. Such a struct has a member named `subscopes`, which is declared as a `union`. Furthermore, the struct contains all variable declarations that are direct children of the scope node. The `subscopes` `union` contains the `struct`s generated from the child scope nodes recursively, by calling the `printAllocationStructure` method with the child scope nodes as a parameter.

```
void printAllocationStructure(ScopeNode n) {
  out.println("struct {");

  for (Iterator it = n.getChildDeclarationNodes().iterator();
            it.hasNext(); ) {
    DeclarationNode dn = (DeclarationNode) it.next();
    out.println(n.getDeclarationAsString()) + ";");
  }

  out.println("union {");
  for (Iterator it = n.getChildScopeNodes().iterator();
            it.hasNext(); ) {
    printAllocationStructure( (ScopeNode) it.next() );
  }
  out.println("} subscopes;");

  out.println("} __scope" + n.getUniqueID() + ";");
}
```

Figure 9: Generation of the allocation data structure.

Code transformation

The TinyVT compiler traverses the scope tree once more, and when visiting a declaration node, it removes the corresponding declaration from the AST, and replaces all references to the variable with the reference to the corresponding member of the generated data structure.

Initially, the `visitNode` method is called with the root of the scope tree and an empty string given as parameters. First, it attaches `__scopeX` to the `path` string, `X` being the unique identifier of the scope node, which represents the prefix of the *path* to the declarations of the scope in the generated data structure. The *path* is a series of enclosing `struct` and `union` identifiers, from the outside inwards, separated with the '`.`' literal.

Then, direct subscopes are traversed iteratively, recursively calling `visitNode` with the node of the subscope and the path concatenated with "`.subscopes.`" as a parameter.[1]

Finally, for every declaration node that is a direct child of the scope node, the corresponding variable declaration is removed from the AST, and references to the variable are changed to reflect the corresponding member of the generated data structure. That is, an identifier that refers to the declared variable, is replaced with the identifier prefixed with the path to the corresponding member within the global allocation structure. For instance, considering the example in Figure 8, references to variable `e` in `scope4` will be replaced with `e` prepended by its *path* string:

[1]`subscopes` is the identifier of the union that holds the `struct`s that correspond to the subscopes.

`__scope1.subscope.__scope2.subscope.__scope4.e`
That is, an expression statement, such as `e = 1.0;` in scope 4 would be rewritten as:
`__scope1.subscope.__scope2.subscope.__scope4.e = 1.0;`

The source code of the `visitNode` function is the following.

```java
void visitNode(ScopeNode n, String path) {
  path = path + "__scope" + n.getUniqueID();

  for( Iterator it = n.getChildScopeNodes().iterator();
            it.hasNext(); ) {
    visitNode( (ScopeNode) it.next(), path + ".subscopes." );
  }

  for( Iterator it = n.getChildDeclarationNodes().iterator();
            it.hasNext(); ) {
    DeclarationNode dn = (DeclarationNode) it.next();
    removeFromAST(dn);
    rewriteReferences(dn, path + "." + dn.getIdentifier());
  }
}
```

Figure 10: Removing variable declarations and rewriting references to point to the generated allocation structure.

Limitations

The prototype TinyVT compiler has several limitations related to compiler-managed automatic variable allocation. The most significant ones are the following:

- First, the algorithm explained above assumes that declarations of automatic variables have no initializers. (A static variable, though, may have an initializer.)
- Second, the types used to declare the automatic variables must be available in the global scope. For example, if an automatic variable is declared as a struct, and the struct is defined elsewhere within the thread (preceding the variable declaration), the declaration cannot be moved to the global scope.
- Types used in automatic variable declarations cannot refer to types that are not available in the global scope. For instance, the declarations
  `double a; int b[sizeof(a)];`
  cannot be refactored by the compiler, since, after moving the declarations to the global scope, references to variables within the declarations are not rewritten.

Nevertheless, the above scenarios can easily be avoided by manual code refactoring in most cases. For example, stripping initializers from declarators, by turning them into assignment statements will solve the first issue, while manually moving definitions of tagged types (`struct`, `union`) to the global scope will solves the second issue. Many, if not all of such refactoring steps can be implemented programmatically. Such transformations, however, are beyond the scope of this work.

2.6.3 The local control flow graph

To enable further code analysis and transformation, the compiler builds the local control flow graph (LCFG) of the thread. Each node in the LCFG represents either a single `await` statement or a set of consecutive statements without any `await` statements, jumps to other nodes or target points of jumps from other nodes. A block of code belonging to a node has exactly one entry point and only one exit. The LCFG is a directed graph: a directed edge between nodes represents a jump in the control flow. There is a dedicated entry block representing the piece of code with which the execution of the thread starts.

In contrast with the typical definition of control flow graphs, the LCFG generated by the TinyVT compiler does not forbid local jumps within the code block of a node, which allows for treating C control structures (`if`, `for`, `while`, etc.) as ordinary statements if they contain no nested `await` statements.

The LCFG only captures *local* control flow: it does not describe the interaction with the thread's environment. Although an `await` statement defines a yield point at which control leaves the thread, as well as an entry point at which thread execution resumes, the control flow graph treats all statements, including `await` statements, opaquely, hiding the fact that control flow is not retained by the thread while the `await` statement is executing. The same holds for function invocation expressions. When the thread calls an external function, it temporarily relinquishes control to the implementation of the function, which, on completion, returns to the thread. Function calls are also opaque: they are handled uniformly with other types of expressions.

Building the LCFG

The TinyVT compiler builds the control flow graph from the abstract syntax tree (AST) which is the output of the variable allocation step. The LCFG is built using a set of recursive functions that visit the nodes of the AST recursively.

The LCFG building algorithm is explained in this section with excerpts (with simplifications) from the compiler source.

The input of the LCFG building algorithm is the abstract syntax tree. Similarly to the scope tree building algorithm, we rely on helper functions to extract information from the AST. The corresponding function names are self explanatory:

- `boolean isAwaitStatement(AST ast)` returns true only if the subtree given as a parameter corresponds to an `await` statement, or,
- `AST getConditionExpression(AST ast)` returns the subtree corresponding to the condition expression of the `if`, `for`, `while`, etc. statement the AST of which is given as a parameter.

The local control flow graph is represented as a set of nodes of type `Node`. Each node object has a unique identifier (a positive integer), a list of statements, an optional conditional expression, and reference to a node (or two nodes) that correspond to edges in the control flow graph. A node with a conditional expression has two references to next nodes: one that specifies the next node that is executed if the expression evaluates to true, and the other for the false branch. Nodes without conditional expressions have only one next node reference. The call graph has a dedicated entry node which is used as a handle to the graph.

The `buildLCFG` method, described below, bootstraps the control flow graph building process. Its only parameter is the AST of the thread definition, and it returns the entry node of the thread's LCFG. The `buildLCFG` method is implemented as follows. First, it creates an empty node which going to be the entry node of the LCFG with its unique identifier explicitly set to 1, which will eventually be returned. Then, it calls the `resolve` method with the AST of the thread's implementation and the entry node as parameters, which will build the rest of the graph and link it to the entry node.

```
Node buildLCFG(AST thread) {
 Node entryNode = new Node(1);
 resolve(getCompoundStatement(thread), entryNode);
 return entryNode;
}
```

Figure 11: The main method of the Local Control Flow Graph builder.

The generic `resolve` method delegates the graph building task to specialized resolve methods (`resolveAwait`, `resolveIf`, `resolveFor`, etc.) based on the kind of statement the AST subtree represents, given as the first parameter. The second parameter specifies the current node of the LCFG to which the code corresponding to the AST subtree will be appended, possibly linking new nodes to it. The resolve method returns the node to which the next siblings of the AST should be linked, i.e. where graph building from the subsequent lines of source code should continue. All specialized resolve methods have identical type signatures, and the semantics of the parameters and return value are the same as with the generic resolve method.

```
Node resolve(AST statement, Node entryNode) {
 if(isAwaitStatement(statement))
  return resolveAwait(statement, entryNode);
 if(isIfStatement(statement))
  return resolveIf(statement, entryNode);
 if(isForStatement(statement))
  return resolveFor(statement, entryNode);
 /* ... and so on, for other statement types*/

 /* finally, default to an expression statement */
 return resolveStatement(statement, entryNode);
}
```

Figure 12: The generic resolve method of the LCFG builder.

If the type of the AST is such that it does not require special handling (e.g. it represents a C statement with no nested statements), the graph building task is delegated to the `resolveStatement` method. It simply adds the statement to the node given as a parameter, and returns the same node.

```
Node resolveStatement(AST statement, Node entryNode) {
 entryNode.add(statement);
 return entryNode;
}
```

Figure 13: Resolving statements.

The LCFG is constructed such that for every `await` statements a new node is created. Therefore, the `resolveAwait` method creates a new node (`awaitNode`), adds the `await` statement to it, and links to it from the entry node. Since `awaitNode` must not contain any other statements, the method creates a new node (`exitNode`) which is linked from the `await` node and then returned. This way, subsequently called resolve methods cannot add C statements to the `await` node, only to the new node that is returned.

```
Node resolveAwait(AST awaitStatement, Node entryNode) {
 Node awaitNode = new Node();
 entryNode.setNext(awaitNode);
 awaitNode.add(awaitStatement);
 Node exitNode = new Node();
 awaitNode.setNext(exitNode);
 return exitNode;
}
```

Figure 14: Resolving the `await` statement.

Below I describe the resolve methods for a representative set of C language constructs: compound statement, `if` statement, `for` statement (with `break` and `continue`) and `goto` statement (with target labeled statements). The methods that resolve other C statement kinds are built using similar techniques, and their description is omitted.

The following method describes how nodes are built from compound statements. If the compound statement contains no `await` statements, no jumps to the outside and no target points of jumps from the outside, it is handled as a simple statement, delegating the graph building task to the `resolveStatement` method. Otherwise, the compound statement must be traversed: the `resolveCompoundStatement` method walks through the direct children of the compound statement's AST, which represent the directly nested statements. The nodes of the LCFG are build recursively by delegating the graph building task of the individual statements to the generic resolve method. Finally, the `resolveCompoundStatement` will return the node that resulted from resolving the last nested statement of the compound statement.

```
Node resolveCompoundStatement(AST compoundStatement, Node entryNode) {
 if(!hasAwaitOrNonLocalJump(compoundStatement)) {
   return resolveStatement(compoundStatement, entryNode);
 } else {
   Node currentNode = entryNode;
   AST currentStatement = getFirstStatement(compoundStatement);
   do {
     currentNode = return resolve(statement, entryNode);
   } while
      ((currentStatement = getNextStatement(currentStatement)) != null)
   return currentNode;
 }
}
```

Figure 15: Resolving compound statements.

As we saw it with the compound statement, an `if-else` statement can be handled as a simple statement if it has no nested `awaits`, jumps to the outside or target points of jumps from the outside. Otherwise, to resolve an `if-else` statement, the `resolveIf` method builds a subgraph with branching, links a common exit node to both branches where they join, and returns the exit node.

First, two new nodes are created for the entry points of the true and false branches. The conditional expression is added to the entry node of the `if-else` statement, which links to the entry node of the true branch if the condition evaluates to true, and to the node of the false branch otherwise. Both branches are built recursively by calling `resolve`, which returns exit nodes of the respective branches. The exit nodes of the branches link to a newly created common exit node, with which the `resolveIf` method eventually returns.

```
Node resolveIf(AST ifStatement, Node entryNode) {
 if(!hasAwaitOrNonLocalJump(ifStatement)) {
   return resolveStatement(ifStatement, entryNode);
 } else {
   Node trueBranchEntryNode = new Node();
   Node falseBranchEntryNode = new Node();
   Node exitNode = new Node();
   entryNode.add(getConditionExpression(ifStatement));
   entryNode.setNextOnTrue(trueBranchEntryNode);
   Node trueBranchExitNode = resolve(
             getTrueBranchStatement(ifStatement),
             trueBranchEntryNode);
   trueBranchExitNode.setNext(exitNode);
   if(getFalseBranchStatement(ifStatement) == null) {
     entryNode.setNextOnFalse(exitNode);
   } else {
     entryNode.setNextOnFalse(falseBranchEntryNode);
     Node falseBranchExitNode = resolve(
             getFalseBranchStatement(ifStatement),
             falseBranchEntryNode);
     falseBranchExitNode.setNext(exitNode);
   }
   return exitNode;
 }
}
```

Figure 16: Resolving the if-else construct.

It is possible that the `else` branch of the if statement is missing. In that case, getFalse-BranchStatement will return `null`, and the the exit node will be linked directly from the entry node if the condition evaluates to false.

Methods building the LCFG subgraph for iteration statements have a structure similar to that of `resolveIf`. There are two additional statements that may be nested in iterations and need special treatment: `continue` and `break`. The LCFG builder must keep track of the target nodes of potential `continue` and break `statements`, and link to them if such statements are encountered.

`Continue` and `break` statements must be the last statements in graph nodes, since they constitute the single allowed exit point of the code within the node. Therefore, the enclosing statement (compound statement, `if-else` statement, etc.) of `break` or `continue` that is nested in the iteration statement being resolved must be broken up into its constituent statements, even if the enclosing statement does not contain any `await` statements.

The code in Fig. 18 illustrates the behavior of the `resolveFor` method. The logic of the resolve methods for other iteration statements (`while`, `do-while`) are very similar, and hence their code is omitted.

Two member variables of the LCFG builder class are declared to hold the target nodes that should be linked to on `continue` and on `break`. Initially they are set to null, as there exists no enclosing iteration statement. If an iteration statement is being traversed, continueNode will refer to the condition testing node, while breakNode to the exit node of the iteration.

```
Node continueNode = null;
Node breakNode = null;
```

Figure 17: Tracking targets of break and continue.

As before, first we check if the for statement needs to be broken up or it can be added to the current node as an opaque statement. If it needs to be broken up, the initialization statement of the for loop is added to the current node, and the following new graph nodes are created: a loop header node containing the loop condition, an entry node for the loop body, and an exit node. The loop header node links to the loop body entry node if the loop condition evaluates to true, and it links to the exit node on false. The exit node of the loop body is computed by calling `resolve` on the statement that constitutes the body of the loop. The exit node of the loop body links back to the loop header node.

While the LCFG builder resolves the body of the loop, it might encounter `continue` or `break` statements. Before calling resolve, the `continueNode` and `breakNode` variables are set to the loop header node and to the exit node, respectively (saving the current values that may belong to an enclosing iteration statement to temporary variables). Later, when a `continue` or `break` is encountered, they will link to the `continueNode` or `breakNode`, respectively. (After the `resolve` method returns, the saved values of continueNode and breakNode are restored.)

`Continue` statements are resolved as follows. The entry node is linked to the node the `continueNode` variable contains, which was previously set by the resolve method that processed the enclosing iteration statement. Since statements that follow the `continue` statement might not be dead code (e.g. labeled statements that are jump targets), the `resolveContinue` method creates an empty exit node and returns it. The exit node is not linked from the continue node, though.

```
Node resolveFor(AST forStatement, Node entryNode) {
 if(!hasAwaitOrNonLocalJump(forStatement)) {
   return resolveStatement(forStatement, entryNode);
 } else {
   entryNode.add(getInitStatement(forStatement));
   Node loopHeaderNode = new Node();
   entryNode.setNext(loopHeaderNode);
   loopHeaderNode.add(getConditionExpression(forStatement));
   Node exitNode = new Node();
   loopHeaderNode.setNextOnFalse(exitNode);
   Node loopBodyEntryNode = new Node();
   loopHeaderNode.setNextOnTrue(loopBodyEntryNode);
   Node savedContinueNode = continueNode;
   Node savedBreakNode = breakNode;
   continueNode = loopHeaderNode;
   breakNode = exitNode;
   Node loopBodyExitNode = resolve(getLoopBodyStatement(forStatement),
           loopBodyEntryNode);
   continueNode = savedContinueNode
   loopBodyExitNode.add(getUpdateStatement(forStatement));
   loopBodyExitNode.setNext(loopHeaderNode);
   return exitNode;
 }
}
```

Figure 18: Resolving `for` loops.

```
Node resolveContinue(AST continueStatement, Node entryNode) {
 entryNode.setNext(continueNode);
 Node exitNode = new Node();
 return exitNode;
}
```

Figure 19: Resolving the `continue` statement.

The method that resolves `break` statements is very similar to `resolveContinue`, except that it links the `entryNode` to the node specified in the `continueNode` variable.

```
Node resolveBreak(AST breakStatement, Node entryNode) {
 entryNode.setNext(continueNode);
 Node exitNode = new Node();
 return exitNode;
}
```

Figure 20: Resolving the `break` statement.

The LCFG builder must resolve `goto` statements the jump targets of which are outside the LCFG node that contains the `goto` statement, since such a `goto` statement constitutes the single allowed exit point of the code within an LCFG node. The `resolveGoto` method unconditionally links from the current entry node a label node, which will link to the actual target statement when the corresponding labeled statement is resolved. It returns a newly created empty node.

```
Node resolveGoto(AST gotoStatement, Node entryNode) {
 Node labelNode = getLabelNode(getLabel(gotoStatement));
 entryNode.setNext(labelNode);
 Node exitNode = new Node();
 return exitNode;
}
```

Figure 21: Resolving the `goto` statement.

The `getLabelNode` helper method maintains a map of label nodes associated to labels. If a node exists in the map associated to the label given as a parameter, it is returned. Otherwise, a new node is created, put in the map, and returned.

```
Node getLabelNode(String label) {
 Node labelNode = (Node)labelNodeMap.get(label);
 if (labelNode == null) {
  labelNode = new Node();
  labelNodeMap.put(label, labelNode);
 }
 return labelNode;
}
```

Figure 22: Retrieving label nodes.

The `resolveLabeledStatement` method creates a new node that will contain the statement (`statementNode`), and links to it from the entry node, and also from the `labelNode` returned by the `getLabelNode` helper function. The statement of the labeled statement is resolved recursively with the `resolve` method (since it may be a statement that must be broken up into its constituent statements because, for instance, it contains an `await`). Finally, the exit node returned by resolving the statement is returned by `resolveLabelStatement`.

```
Node resolveLabeledStatement(AST labeledStatement, Node entryNode) {
 Node statementNode = new Node();
 entryNode.setNext(statementNode);
 Node labelNode = getLabelNode(getLabel(labeledStatement));
 labelNode.setNext(statementNode);
 Node exitNode = resolveStatement(getStatement(labeledStatement),
             statementNode);
 return exitNode;
}
```

Figure 23: Resolving labeled statements.

Once the entire thread definition is resolved, the final step of the LCFG builder is simplifying and pruning the graph. Nodes A and B are merged if neither contains an `await` statement, there is a directed edge from A to B and no other edge leads to node B. Finally, since the `buildLCFG` method only returns the entry node of the thread, nodes not reachable from the entry node are automatically removed from the graph.

Identification of yield points

Execution of a TinyVT thread progresses in discrete uninterrupted steps. On an external event, a thread resumes execution with the statement at which the thread last yielded. The thread, therefore, must keep track of the location of the last yield point, that is, the identifier of the last yield point must be stored as part of the thread's state. For further code generation, it is required that the compiler computes the size of the state space, and that it identifies all possible yield points.

Yield points are explicitly described in the source code of a thread: TinyVT provides language constructs to specify yield points, and, in the same time, it guarantees that the thread does not yield anywhere else. Since the syntactic normalization step translated all `yield` and `ireturn` statements to `await` statements, all of which are wrapped in special `await` nodes of the local control flow graph, identifying yield points requires enumerating the `await` nodes of the LCFG. The compiler assigns consecutive positive integers $1..n$ (unique identifiers of yield points) to `await` nodes of the LCFG, and remembers the highest number assigned. The number of yield points is stored in the `yieldPointCount` variable. This information will later be used by the code generator.

Since there is a direct mapping between yield points and `await` statements, the terms *identifier of a yield point* and *identifier of an `await` statement* can be used interchangeably. It is important to note that they are different from the identifiers of graph nodes, however.

Enumeration of input events

When thread execution resumes in response to an external event, control is passed to an event handler inlined in the `await` statement that corresponds to the current thread state. Since multiple `await` statements may contain handlers of the same kind of event, control must pass to the thread through a compiler-generated common event handler stub, which dispatches the inlined event handlers after inspecting the thread state. In order to generate these event handler stubs, the compiler inspects all `await` nodes to construct the set of all event kinds (i.e. all unique function signatures of inlined event handlers), and builds a data structure that assigns to each event kind the identifier of the `await` node which contains a corresponding handler.

2.6.4 Code generation

The code generation step of the compiler consists of four parts. First, the code that implements querying and altering the thread state is generated. Then, for every node of the local control flow graph, the compiler outputs a function that contains the code within the node (referred to as a *block function*). In the third step, every inlined event handler is turned into a function definition. Finally, event handler stubs are generated for every event type the thread may accept, which dispatch the functions generated from the inlined event handlers depending on the current thread state.

Thread state and common functions

The thread state is stored either in an eight-bit-wide or in a sixteen-bit-wide unsigned integer variable. The compiler computes the number of bits required to store the state with the formula

$$stateBits = log2(yieldPointCount) + 2.$$

The two extra bits are used for guarding the `yield` statement generated from `ireturn` statements, and for testing for reentrance violations. If `stateBits` is less than or equal to eight, the `state_t` type will be defined as `uint8_t`, otherwise as `uint16_t`. The types `uint8_t` and `uint16_t` are defined in the C99 standard `stdint.h` file.

The compiler generates the following state related code. The thread state is stored in the `__state` variable, the two least significant bits of which are the two flags, while the rest of the bits store the last yield point.

```
__state_t __state;
```

Figure 24: Storage of thread state. The `__state_t` type is defined either as `uint8_t` or as `uint16_t`, depending on the number of await nodes in the LCFG.

Helper functions are generated to set, clear and check the flags, as well as to get and set the last yield point. The `bool` type is defined in the C99 standard `stdbool.h` file.

```
enum { __EXECUTING_MASK = 1,
       __YIELD_AFTER_AWAIT_MASK = 2 };

inline bool __is_set_flag(__state_t flag_mask)
              { return __state & flag_mask; }
inline void __set_flag(__state_t flag_mask)
              { __state |= flag_mask; }
inline void __clear_flag(__state_t flag_mask)
              { __state &= ~flag_mask; }

typedef __state_t __yield_point_t;
inline __yield_point_t __get_yield_point() { return __state >> 2; }
inline void __set_yield_point(__yield_point_t yp) {
 __state = (yp << 2) | (__state & 3); }
```

Figure 25: Helper functions to manage thread state.

Generating code from LCFG nodes

Nodes of the control flow graph are turned into C functions, where the unique identifier of the node is encoded in the function name (e.g. `__block0` or `__block5` for nodes with unique id 0 or 5, respectively). The block functions have an empty argument list, and return the identifier of integer type `__next_block_t` of the block that should be executed next. It will always be the identifier of a node linked from the current node. If a node contains a conditional expression, the id of the next block is chosen runtime, after evaluating the expression. The block functions generated from `await` blocks always return a reserved constant, `__YIELD_BLOCK`, causing the thread to yield (no subsequent blocks will be executed).

First, the `typedef` for the block identifiers, and the definition of the `__YIELD_BLOCK` constant are created.

```
typedef uint16_t __next_block_t;
enum {__YIELD_BLOCK = 0};
```

Figure 26: Return type of the block functions.

Then, block function definitions are generated. There are three different kinds of graph nodes, resulting in different code:

- **Non-await nodes without branching.** The generated block function contains the list of statements within the node, and returns the identifier of the node to which there is an out-edge in the LCFG (i.e. the node that should be executed next). The example below is generated from node with identifier 1. It contains one statement (a printf), and returns the identifier of the next node.

```
__next_block_t __block1() {
  printf("__block1\n");
  return 2;
}
```

Figure 27: Block function of a simple non-await node.

- **Non-await nodes with branching.** Such nodes have two out-edges and contain a conditional expression. The generated function consists of the list of statements in the node, followed by returning the identifier of the next node, which is chosen runtime by evaluating the condition expression. The example code below is generated from node 2, which links to node 3 if its conditional expression evaluates to true, and to node 5 otherwise. It contains one statement, followed by evaluating a conditional expression, and returns the appropriate node identifier.

```
__next_block_t __block2() {
  printf("__block2\n");
  if(isReady)
    return 3;
  else
    return 5;
}
```

Figure 28: Block function of a non-await node with branching.

- **Await nodes.** The block functions that are generated from `await` nodes contain the code which is executed when the thread yields on encountering the `await` statement. The generated code will save the identifier of the yield point that corresponds to the `await` statement in the thread state. The identifier of the yield point had previously been assigned to the `await` statement when the yield points were enumerated in a previous compiler pass. Since no more statements will be executed within this execution step, the block function returns the `__YIELD_BLOCK` constant, from which the function's caller will know that the thread needs to yield.

```
__next_block_t __block4() {
  __set_yield_point(2);
  return __YIELD_BLOCK;
}
```

Figure 29: Block function of an await node.

Inlined event handlers

The TinyVT compiler generates a separate function for every inlined event handler. The signature (specifiers, return type, and argument list) of the generated function will be identical to that of the inlined event handler, and the name of the generated function is mangled to include the identifier of the yield point associated with the enclosing `await` statement. This mangling is required since multiple `await` statements may block on the same event kind. However, they may have different event handler implementations inlined. The generated event handler, after executing the inlined code, will call the block function of the node that is linked from the `await` node that holds the enclosing `await` statement. The block function will return the identifier of the next node, and the next node's block function will be called next. This way, block functions are dispatched iteratively until a `__YIELD_BLOCK` is returned, which causes the inlined event handler to return.

Before generating the functions for the inlined event handlers, the compiler generates the helper function `__dispatch_next_block`, which dispatches a block function based on the block function identifier, given as a parameter. The helper function returns the id of the block that should be executed next, or the `__YIELD_BLOCK` constant, if the thread should yield. Since the name of the block function is mangled such that it ends with the identifier of the block, the lines containing "`case X: return __blockX();`" can be generated in an iteration over all LCFG nodes, writing the unique identifier of the node in place of `X`. The code below was generated from an LCFG with six nodes.

```
__next_block_t __dispatch_next_block(__next_block_t next___block) {
  switch (next___block) {
    case 0: return __block0();
    case 1: return __block1();
    case 2: return __block2();
    case 3: return __block3();
    case 4: return __block4();
    case 5: return __block5();
    default: return __YIELD_BLOCK;
  }
}
```

Figure 30: Dispatching logic for block functions.

Below is an example of a function generated from the `myEvent` event handler inlined in the `await` statement with unique identifier 1. Notice that the compiler generates the declaration of the `__rval` variable with the appropriate type. Lines 3 to 6 contain the body of the inlined event handler, which is followed by the code that sets the next block and iteratively dispatches the block functions until `__YIELD_BLOCK` is returned. In preparation for yielding, the flag indicating that the thread is executing is cleared. Finally, the function returns with the value stored in the temporary `__rval` variable.

```c
char __myEvent_await1() {
  char __rval;
  { printf("await1/myEvent\n");
    __rval = 'a';
  }
  { __next_block_t __next_block = 1;
    while((__next_block = __dispatch_next_block(__next_block))
             != __YIELD_BLOCK);
  }
  __clear_flag(__EXECUTING_MASK);
  return __rval;
}
```

Figure 31: Example of a function generated from an inlined event handler.

A dedicated event handler, the name of which is mangled to include "`_halt`" after the event name, is also generated for every event kind the thread reacts to. This handler is called if a runtime error occurs, when the thread cannot handle the received event. The default implementation of these error handlers halts the execution of the program. Although the function does have a return statement if the function's return type is non void, control never reaches it. The implementation of the `halt()` function is specific to the execution environment (e.g. `exit()` on POSIX, or `while(1)` in TinyOS). (In the prototype TinyVT compiler, the return expression is chosen arbitrarily to be a recursive function call, because it is the easiest to generate, since this way the return statement can be generated without inspecting the return type.)

```c
char myEvent_halt() {
  halt();
  return myEvent_halt();
}
```

Figure 32: Example of a generated error handler.

Event handler stubs

For every event kind the thread reacts to, an event handler stub is generated that matches the function signature of the corresponding inlined event handler(s). Such an event handler stub includes a `switch` control structure, which dispatches the event handler generated from the `await` node the identifier of which matches that of the the current yield point. A runtime error occurs if either the thread is currently executing a step (reentrance violation), or if the received event is not accepted in the current state.

To generate the event handler stub, the compiler uses the data structure, built when enumerating the input events, that assigns to each event kind (signature of event handler) the identifiers of `await` statements that contain a handler of that event kind inlined. For every event kind, an event handler stub, such as the one below, is generated. For every `await` statement that includes a handler of the event, a line in the form of "`case X: return __EVENTNAME_awaitX();`" is added to the `switch` construct, where `EVENTNAME` and `X` are replaced with the name of the event and the identifier of the `await` statement's yield point, respectively.

```
char myEvent() {
  if(__is_set_flag(__EXECUTING_MASK))
    return myEvent_halt();
  __set_flag(__EXECUTING_MASK);
  switch(__get_yield_point()) {
    case 1: return __myEvent_await1();
    case 2: return __myEvent_await2();
    default: return myEvent_halt();
  }
}
```

Figure 33: Example of a generated event handler stub.

For the implicit initialization event, the compiler generates the following code, which bootstraps the thread named `myThread`, by executing all statements up to the first yield point. Initially, the flag indicating that the thread is executing is set. Then, starting from the block function with identifier 1, which corresponds to the entry node of the thread, block functions are dispatched iteratively until `__YIELD_BLOCK` is returned, indicating that the first yield point is reached. Finally, the flag indicating that the thread is executing is cleared, and the `myThread` function returns.

```
void myThread() {
  __next_block_t __next_block = 1;
  __set_flag(__EXECUTING_MASK);
  while((__next_block = __dispatch_next_block(__next_block))
             != __YIELD_BLOCK);
  __clear_flag(__EXECUTING_MASK);
}
```

Figure 34: Generated thread initialization code.

For reference, a simple TinyVT application, along with the corresponding compiler output is given in Appendix A.

2.7 Case study

To illustrate the expressiveness of the TinyVT through three examples. First, I show that the I2C packet-level interface, used as a motivating example in Section 2.2. Then, I present the TinyVT implementation of the main component of TinyOS's Surge application. Surge is a simple sensing program that collects sensor readings and forwards them to a sink node using an underlying routing service. Since it captures an important aspect of WSN applications, Surge, or a similar sense-and-forward algorithm is often used as a benchmark application in the literature [8, 1, 27, 40, 47, 55, 56, 60, 62]. Finally, I present a simple packet forwarding service that accepts messages from the application layer as well as from the radio driver, and sends them to the parent node in a multihop routing topology.

2.7.1 I2C packet-level interface

We illustrate the expressiveness of TinyVT by rewriting the I2C packet-level interface example, described previously in Section 2.2, using the thread abstraction.

Below I present the source code of the `i2c_writepacket` thread. Notice how this code resembles the pseudocode presented Section 2.2.

In the idle state, i.e. when no client request is being processed, the thread blocks on the `i2cpacket_write` command. If a client request comes in, the inlined implementation of the command is executed, requesting access to the I2C bus by calling the `i2c_sendStart` command. The thread blocks as the next await statement is reached. Once access to the bus is granted, the underlying byte-level I2C service invokes the `i2c_sendStartDone` callback function. This manifests itself in the thread as an occurrence of the `i2c_sendStartDone` event, which resumes the the execution of the thread. 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.

The packet is written out byte by byte to the bus, waiting for an `i2c_writeDone` callback after each `i2c_write` request. Finally, the thread requests releasing of the bus by issuing the `i2c_sendEnd` call and blocks until the `i2c_sendEndDone` occurs. After the I2C bus is released, completion of packet transmission is reported to the client by invoking the client's `i2cpacket_writeDone` callback function.

This algorithm is running in an infinite service loop, hence, once a packet transmission is complete, the service is reset to the idle state, awaiting new packet transmission requests.

```
thread i2c_writepacket {
  while(1) {
    uint8_t *packet_data, packet_length;

    await( void i2cpacket_write(uint8_t length, uint8_t* data) {
             packet_data = data;
             packet_length = length;
             i2c_sendStart();
             dreturn;
           }
    );

    await( void i2c_sendStartDone() { dreturn; }
    );

    {
      uint8_t index;
      for(index=0; index<packet_length; ++index) {
        i2c_write(packet_data[index]);
        await( void i2c_writeDone() { dreturn; }
        );
      } /* end for */
    }

    i2c_sendEnd();
    await( void i2c_sendEndDone() { dreturn; }
    );

    i2cpacket_writeDone();
  } /* end while */
} /* end thread */
```

Figure 35: Packet-oriented I2C driver in TinyVT. Notice the resemblance with the pseudocode presented in Fig. 1.

2.7.2 The Surge application

Surge is a simple sense-and-forward data collection application, a TinyOS based implementation of which is publicly available in the TinyOS source code repository at sourceforge.net. Driven by a periodic timer, Surge samples the ADC to acquire a sensor reading, wraps it in a data packet and hands it over to the routing service which will forward it to a designated sink node in a multihop topology.

The corresponding TinyVT implementation is rather simple. First, the timer is started with period `TIMER_RATE`. Then, control enters a loop in which sensor readings are acquired and transmitted. The execution of the code within the loop is triggered by the `timer_fired` event, a callback from the timer service. In response to the fired event, the thread requests a sensor reading from the ADC subsystem and blocks until the sample is acquired. Acquisition is reported by the `adc_dataReady` callback from the ADC module, which provides the read value as a parameter. In the corresponding event handler, the sensor reading is written to the message packet and the length of the packet's payload is set to 2 (the size of the `uint16_t` type).

Then, the packet is handed over to the routing service by calling the `multihop_send` function. However, since the routing service might not be able to accept the request (it can be in a busy state), the request might return FAIL. Therefore, the thread keeps retrying sending the packet until it is accepted by the routing layer. Notice that the corresponding while loop contains the yield statement. The yield statement passes the control to the dispatcher in the event-driven runtime, which may schedule other services (e.g. the routing service or the radio stack) before continuing the execution of the current thread.

Once the routing layer accepted the packet, the thread blocks on the `multihop_sendDone` event, a callback signaling that the packet has been transmitted over the radio. After that, control returns to the beginning of the body of the outer loop, blocking on the next fired event from the timer.

```
thread surge {
  timer_start(TIMER_REPEAT, TIMER_RATE);

  while(1) {
    Msg msg;

    await( void timer_fired() {
              adc_getData();
              dreturn;
           }
    );

    await( void adc_dataReady(uint16_t reading) {
              (uint16_t*)msg.data)[0] = reading;
              msg.length = 2;
              dreturn;
           }
    );

    while( multihop_send(&msg) != SUCCESS) {
      yield;
    } // end while

    await( void multihop_sendDone() {
             dreturn;
           }
    );
  } // end while
} // end thread
```

Figure 36: The surge application in TinyVT.

2.7.3 A simple multihop packet forwarding engine

The third example presented in this section illustrates how TinyVT can be used to implement a simple multihop packet forwarding service. It provides a `multihop_send` service function to the clients, and reports the completion of the packet transmission via the `multihop_sendDone` callback. The actual transmission of the packet is delegated to the underlying radio stack, which provides a `radio_send` service function and signals completion via the `radio_sendDone` callback. The multihop service is also forwarding packets it received over the radio. The radio stack indicates packet reception bye invoking the `radio_receive` handler.

```
thread multihop {
  Msg msg, *msgPtr1;
  msgPtr1 = &msg;

  while(1) {
    Msg *msgPtr2;
    msgPtr2 = NULL;

    await( Msg* radio_receive(Msg* m) {
             Msg* tmpMsgPtr = msgPtr;
             msgPtr1 = m;
             dreturn tmpMsgPtr;
           }
           result_t multihop_send(Msg* m) {
             msgPtr2 = msgPtr1;
             msgPtr1 = m;
             dreturn SUCCESS;
           }
    );

    while( radio_send(parent_address(), msgPtr1) != SUCCESS) {
      // yield;
      dpc_request(&deferred_proc);
      await ( void deferred_proc () { dreturn; }
               Msg* radio_receive(Msg* m) {
                 dpc_cancel(&deferred_proc);
                 dreturn m;
               }
          );
    } // end while

    await( void radio_sendDone(Msg* m) {
             if(msgPtr2 != NULL) {
               multihop_sendDone();
               msgPtr1 = msgPtr2;
             }
             dreturn;
           }
           Msg* radio_receive(Msg* m) { dreturn m; }
    );
  } // end while
} // end thread
```

Figure 37: Multihop packet forwarding in TinyVT.

The corresponding TinyVT implementation (Fig. 37) runs a service loop, in which, the thread first initializes the `msgPtr2` variable to NULL and then it waits for a `radio_receive` or a `multihop_send` event. Notice that the corresponding await statement on line 9 has two inlined event handlers. The occurence of either of these events resumes the execution of the thread. On a `radio_receive` event, the pointer to the received packet is stored in the `msgPtr1` local variable, and the pointer to an unused message structure is passed back to the radio stack. If a `multihop_send` event occurs, the pointer to the unused packet is saved to `msgPtr2`, and the pointer to the client's packet is saved in `msgPtr2`. After the await statement, `msgPtr1` points to the packet to be sent.

The while loop on line 21 requests the transmission of the packet from the radio stack. Similarly to the previous example, the service returns FAIL when busy, therefore the thread keeps repeating the `radio_send` call until the packet is accepted by the radio stack. The thread must yield in the loop body to allow other events to be dispatched between consecutive retries. However, TinyVT's yield statement is not safe to use in this situation, because a new radio packet might be received in the meantime, and such event must be handled. Therefore, the loop body contains a deferred procedure call (DPC) request, and blocks on either the the DPC callback or a message reception event. In the case when a `radio_receive` event occurs before the DPC is serviced, the received packet is dropped and the DPC request is canceled.

Once the packet is accepted by the radio stack, the thread blocks on the completion event of the transmission (and also on message reception, in which case the received packet is discarded). When transmission is complete, the thread decides if the transmitted packet came from the client or from the radio, by checking the value of `msgPtr2` (`radio_receive` leaves it NULL, while `multihop_send` uses it as temporary storage). In the former case, the `multihop_sendDone` callback of the client is invoked. Finally, control returns to the beginning of the service loop and the thread is ready to accept the next packet.

2.8 Discussion

TinyVT's thread abstraction is a tool that allows for intuitively expressing computation in event-driven systems. The abstraction provided by the language enables the programmer to describe the control flow of a service as if the service had its own, dedicated thread of execution. The source code of TinyVT programs, which may contain multiple threads and arbitrary C code, is translated by the TinyVT compiler to C code, which runs on top of a simple event-driven runtime. The compiler's task is to bridge the large semantic gap between the TinyVT code that relies on a thread abstraction and the resulting C code, where threads are resolved to a set of related event handlers and declarations representing local thread state.

However, all abstractions come at a cost. Below, I investigate the advantages and disadvantages of TinyVT over the traditional multithreading model, as well as event-oriented programming, with respect to functionality, computational overhead and memory usage.

2.8.1 TinyVT versus multithreading

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 TinyVT thread. It is compiled to a set of event handlers, each of which run in the context of its caller, and use the caller's stack to store local variables and function invocation related data such as parameters, return address, registers, etc.

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 compile time.

The event-driven code generated by the TinyVT compiler 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.

TinyVT threads are driven by interaction with their environment. Although a TinyVT thread is programmed assuming an independent thread of execution, it requires a series of external stimuli – either from the underlying event-driven runtime, or from other threads – to trigger thread execution. Since TinyVT threads are compiled to a set of event handlers, implemented as C function definitions, these stimuli are simple function invocations.

Since TinyVT source code is translated to C, the threading abstraction is hardware independent. Unlike operating system kernels or user-space threading libraries, which must be implemented in a platform-specific way (often programmed in assembly), TinyVT does not have to be ported. TinyVT threads are portable as long as there exists a C compiler for the target platform, and an event-driven runtime is available with the assumptions described in Section 2.4.2.

Functionality

Unlike preemptive multithreading, TinyVT threads are not preemptible. Functionally, TinyVT's thread abstraction is closer to that of cooperative multithreading, but more limited in the following respects:

- **TinyVT threads are static.** Unlike in traditional multithreading, where threads must be programmatically spawned and can be explicitly cancelled, TinyVT threads are static. This means that a thread is automatically instantiated after the program is loaded, and ready to accept events from the environment. A TinyVT thread never exits. It either runs in an infinite loop, or is permanently blocking after the end of the control flow is reached (at the implicit empty "`await();`" statement).

- **No built-in IPC mechanisms.** In a cooperative multithreaded programming environment, threads synchronize and communicate using inter-process communication (IPC) mechanisms, such as signals or mutexes. IPC mechanisms are implemented by the runtime (kernel or threading library): for example, when a thread sends a signal to another, the runtime may choose to block the sender thread and schedule a third thread of higher priority than the recipient of the signal, deferring signal delivery.

  TinyVT provides no language support for POSIX-like IPC mechanisms. This has two important implications. First, TinyVT threads can communicate directly with each other via function calls, without going through the runtime. Second, as the runtime (i.e. the dispatcher) does not intercept thread-to-thread calls, the caller thread explicitly defines which thread will be executing next, therefore, the runtime does not have control over this.

- **No thread priorities.** In traditional multithreading, the scheduler chooses which threads to run of those that are ready by inspecting the thread priorities. TinyVT offers no such functionality, since the event-driven runtime is not aware of the tread abstraction. The event dispatcher invokes the event handlers that are generated by the TinyVT compiler from the TinyVT source code.

However, priority aware event dispatching can be used to mimic priority based scheduling schemes of multithreading systems. While TinyVT does not provide language support for assigning priorities to TinyVT threads, nothing prevents the programmer to assign priorities to events that a thread accepts. Instead of setting the priority of a thread to N, the scheduler – implemented, for example, using a priority queue – should be configured such that it assigns priority N to all events designated to the thread. A discussion of such a scheduler implementation is beyond the scope of this work, though.

Performance

The overhead associated with the thread abstraction in traditional multithreading systems originates from context switching. Context switching between threads is usually computationally expensive: it involves the saving of registers, stack and instruction pointers, and other thread specific control structures of one thread, and restoring those of the newly scheduled thread afterwards. In case of preemptive multithreading, context switching is carried out by the operating system kernel, while in cooperative multithreading, this task can optionally be outsourced to a threading library.

TinyVT effectively avoids the need for context switching in the traditional sense. When a TinyVT thread resumes, it is executing using the context (i.e. the stack) of the triggering event. Since all events – directly or indirectly – originate from the event dispatcher, the whole system can use a single stack. A context switch in the TinyVT sense is just a function call, where some registers that are used by the caller need to be saved temporarily for the time of the function invocation, and restored thereafter. This is very inexpensive computationally, commonly resulting in no more than a few dozens of machine instructions. Since the C compiler is aware of the whole static call flow graph within the same translation unit, the compiler can optimize register allocation or automatically inline functions, which results in drastically decreasing (or even eliminating) the cost of TinyVT context switches.

RAM usage

On wireless sensor nodes, RAM is a precious resource. Sensor nodes are typically equipped with only a few kilobytes of RAM, which holds both the stack and statically allocated variables. Typically, there is no heap, since dynamic memory management is not used. This is primarily because the overhead associated with dynamic memory allocation, which is mostly due to fragmentation, is prohibitive on platforms with such small amount of memory.

Stack

One of the often touted disadvantages of multithreading operating systems for sensor nodes is their excessive memory requirements. Each thread requires a dedicated stack, where, when the thread is suspended, the machine state corresponding to the thread is saved. Also, the thread's stack is used for storing automatic local variables, as well as for passing function parameters and return values. As a result, the number of concurrent threads is drastically limited in such systems. For instance, the MANTIS operating system running on the Berkeley MICA2 mote cannot have more than six threads active at a time [8].

TinyVT avoids this problem by assuming an event-driven runtime with a single stack, which is unrolled every time an event handler, dispatched by the runtime, completes. Since event handlers cannot be preempted before they complete, the maximum stack usage of the whole system is the maximum of the stack usage of the individual event handlers.

Statically allocated memory

TinyVT stores the thread state (the identifier of the last yield point and two Boolean flags) in static memory. Depending on the number of yield points within a thread, thread state occupies as little as one or two bytes.

In multithreading, static memory contains global variables and variables designated with the "`static`" storage class specifier. TinyVT, in addition, allocates compiler-managed automatic variables in static memory. The amount of static memory required for their storage equals to the sum of the sizes of compiler-managed variables which may be active concurrently (i.e. those with overlapping scopes). Notice that, in multithreading systems, such variables are stored on the stack. That is, TinyVT, in fact, trades stack space for static memory.

Depending on their placement within the code, TinyVT's `yield` statements may be very expensive in terms of memory usage. The allocation of local non-static variables within a compound statement that contains a blocking wait (yield or await) is managed by the compiler, therefore, it is suggested that yielding be avoided if possible where large local data structures are declared. One important feature of TinyVT is that yield points are explicit, and thus, the programmer has complete control over which variables will be subject to compiler-managed or C's native stack based automatic variable allocation.

2.8.2 TinyVT versus event-oriented programming

Functionality

Pure event-driven programs – where the term *pure* refers to the constraint that all event invocations, directly or indirectly, must originate from the single-threaded event dispatcher – can always be implemented as TinyVT threads if they are free from recursive event handlers.[2] The simplest way to achieve this would be wrapping each handler of the event-driven program in a separate await statement, placed in an infinite loop within a TinyVT thread.

The TinyVT language allows for combining standard C code and TinyVT threads within the same translation unit. Therefore, if the module is such that using TinyVT does not offer any benefits, programming event handlers as C functions is preferred. TinyVT is not a silver bullet. It is widely known that not all patterns of control flow can be conveniently expressed in a thread-like fashion. Nevertheless, the programmer can always fall back to using plain event-driven C code in such cases, and write TinyVT threads only when it is convenient.

Performance

Since TinyVT threads are translated to a set of event handlers by the TinyVT compiler, the generated code will never be better than the *best* hand-written code with the same functionality. Every time an event is dispatched to a thread, the generated code checks for non-reentrance violation and reads the current thread state to check which handler implementation should be executed. Before the event handler returns, the generated code updates the thread state. The corresponding instructions, typically not more than ten, constitute a performance overhead if the event should always be accepted irrespective of the thread state, or, in the latter case, if the thread state does not change in response to the event.

In a typical use case for TinyVT, events trigger in different actions depending on the thread's local state. In such a case, the state and flag checks and updates described above, must also be done in the corresponding hand-written code. As the complexity of the program increases, however, manual control flow management becomes harder. This is where the use of TinyVT pays off, since the thread abstraction can relieve the programmer of the burden of implementing the module as an explicit state machine.

The generated code involves a number of C functions, a series of which is executed in response to a single triggering event. However, an optimizing C compiler, that carries out constant propagation and automatic inlining, can inline most of these functions, thereby eliminating (or drastically reducing) the corresponding performance overhead.

[2]Recursive event handlers are rarely used in event-driven systems on memory-constrained platforms, as recursion, in general, is considered "harmful" because of potentially extensive stack growth.

RAM usage

TinyVT can be thought of as an extension of the event-driven paradigm where the action in response to an event depends not only on the event kind, but also on the local state of the module. In order to dispatch event handlers based on event kind *and* local state, the C code generated from the thread maintains the threads local state.

The TinyVT compiler typically allocates one byte per thread in static RAM to hold the thread state (two bytes if the number of yield points is more than 63.) Hand written modules, in which manual control flow management is required, also use at least one byte for this purpose, therefore, TinyVT's thread state variable typically does not contribute to the memory usage overhead.

TinyVT's most important asset with respect to memory usage is the compiler-managed allocation of local variables with C's automatic storage duration semantics. The allocator algorithm uses the nesting of scopes in TinyVT threads to find out which variables are never active at the same time, since it is always safe to allocate such variables to the same memory area. Manually creating such an allocation in hand written code is a very tedious and time consuming task. Manual allocation, however, may result in better memory allocation because the programmer has better knowledge on variable lifetime that what the compiler can extract from the nesting of scopes. However, as programs evolve (e.g. new features are added or old ones are removed), even small changes to the program logic may require a complete overhaul of the allocation, which drastically increases the maintenance effort. The most important advantage of TinyVT's compiler-managed memory allocation feature is that it relieves the programmer from this complex and tedious task.

2.8.3 Applicability

Overall, TinyVT is best suited to replace the traditional (pure) event-oriented approach if the program's control flow is reasonably complex but it is natural to describe using C control structures. In such use cases, TinyVT can take over the management of local automatic variables from the programmer, which results in better static memory usage than declaring them as global or static, which is the common programming practice in event-driven systems.

2.8.4 Limitations

Asynchronous events

TinyVT threads are assumed to execute on top of a *pure* event-driven runtime, in which only the event dispatcher may call into the event handlers. The dispatcher is assumed to be single threaded, meaning that at most one event handler may be executing in the system at a time. Specifically, interrupt handlers, or other external threads of execution may not invoke the event handlers directly: such calls must go through, and be serialized by, the event dispatcher. This assumption is essential, since the atomicity of event handler executions cannot be guaranteed otherwise, and race conditions could occur.

Some event driven operating systems (e.g. Contiki [22] or TinyOS [44]), however, do not forbid asynchronous invocation contexts to propagate into event handler code. The rationale for this is that the operating system does not need to have a well-defined kernel this way: device driver code and application code can be handled uniformly. Also, this approach allows timely response to interrupts, even in high-level components well above the hardware-software boundary.

TinyVT is not particularly well suited for such *non-pure* event-driven systems. Since threads are guarded against reentrance, an event is only accepted if the thread is blocked. Even if an asynchronous event and a dispatcher-invoked event which it interrupts never access the same set of variables, TinyVT disallows the asynchronous event, since there are potential race conditions in the compiler-generated code. In particular, the dispatcher-invoked event would set the next state of the thread to some value on completion, however, the asynchronous event might set the next thread state to a different value. Typically, the former would win, however, it is possible that the next thread state, which is represented on two bytes, is set to an inconsistent value if the dispatcher-invoked handler is interrupted after the first, but before the second byte is written.

Nevertheless, TinyVT threads *can* be used in such systems. The programmer, however, must make sure that no events arrive while the thread is executing, only when the thread is blocked.

Access to the context of the triggering event

Currently, TinyVT does not support accessing local variables declared within inlined event handlers from the code that follows the enclosing await statement. This seems natural, since such access would violate C's scoping rules. However, the code following the await statement is always executed within the context of the triggering event, and variables declared in the event handler are still alive on the stack until the next yield point is reached.

Therefore, a possible enhancement of the TinyVT language could include a feature to support sharing data between the inlined event handlers and the code following the enclosing await using the stack. This would reduce the static RAM requirements of TinyVT programs, because now, variable sharing is only possible through global, static or compiler-managed local variables. One possible solution for this would be providing a new TinyVT keyword to access the data associated with the current invocation context, containing kind of the triggering event, its parameters, return value (the `__rval` variable) and variables declared locally in the inlined event handler.

Whole-program analysis

The prototype implementation of the TinyVT compiler processes each thread separately. In many cases, however, better memory usage would be achievable through whole-program analysis. For example, if the state of a thread can be represented using only four bits, another four bits are wasted in the byte that is allocated for the thread state. To prevent this, thread state handling could be factored out to a common component, which allocates a variable to hold the global system state, and provides thread-specific state accessor and mutator functions.

Also, whole-program analysis could improve compiler-assisted memory management by identifying scopes in different threads with non-overlapping lifetimes. This would result in better static memory usage, however, computing such allocation is a nontrivial task, and requires that the compiler build and analyze the global control flow graph.

Compiler-assisted memory management

Currently, the TinyVT compiler computes the allocation of compiler-managed variables using the information on the nesting of their scopes. However, in many cases, a variable does not need to stay active until the control exits the compound statement in which the variable is declared: it only needs to be active until it is last accessed. Fine-grained knowledge if variable usage patterns would enable more aggressive compiler-assisted variable allocation strategies, since the compiler could have a more fine-grained picture of the usage patterns of the variables.

One way of achieving this is through compile-time functions, such as `alloc()` and `free()`, where `alloc()` must be called before the first use of the variable, and `free()` would inform the compiler that the variable is not needed any more. These functions would be completely resolved by the compiler and would not appear in the generated C code. In fact, compile-time functions are just a form of annotations that are used to control the behavior of the compiler.

Such a feature would improve on the static memory usage of TinyVT programs. Nevertheless, code that is as good as or better than the memory allocation code generated by the prototype TinyVT compiler is already very hard to produce manually.

Chapter III

Semantics

3.1 Background and related work

Writing and comprehending computer programs requires understanding the precise meaning of the constructs of the programming language used. For many languages, while the syntax is properly specified, semantics is described informally, typically in a natural language (e.g. in English), with examples of code and the description of the expected behavior of a computer that executes it. Unfortunately, such textual specification of semantics can be unintentionally ambiguous, if not misleading. A good example of this is the word ”or” in a natural language, which, without additional disambiguation, can mean disjunction, but exclusive disjunction as well. Semantic ambiguities can lead to incorrect software, as programmers and development tools may have different, conflicting assumptions about the vaguely defined semantics. Clearly, formal specification of semantics for a programming language is of key importance.

Before immensing into the review of various approaches to formal specification of semantics, some terms and concepts that are essential to understand these approaches need to be described.

3.1.1 Syntax

The syntax of a programming language defines the well-formed sentences that can be described in the language. Syntax is only concerned with form and structure, not with the underlying meaning of programs. For textual languages, syntax defines how input symbols (characters) are used to form valid sentences (programs). For graphical languages, syntax defines the elements of the language (graphical artifacts such as shapes, connections, textual annotations, etc.) and the set of rules specifying what configurations of those are valid.

Concrete syntax

The concrete syntax of a programming language is concerned with representation, that is, how programs are expressed as linear streams of characters or, for graphical languages, as sets of two-dimensional graphical objects. The concrete syntax of a textual language can be specified in terms of production rules, for example using the Backus-Naur (BNF) notation . The set of production rules that unambiguously describe which sentences are syntactically well-formed elements of the language is called the grammar of the language.

Abstract syntax

The abstract syntax, on the other hand, is concerned exclusively with the structure of the language, focused around relations between language elements, such as, for example, hierarchy and sequentiality. The conversion from concrete syntax to abstract syntax is called parsing. Parsing includes reading a linear stream of input symbols and transforming it into a tree, called the abstract syntax tree (AST). A common practice is that nonterminals representing operations will become roots of subtrees in an AST, and their children will be the subphrases corresponding to the operands. The AST is an unambiguous, abstract representation of a well-formed program, which reveals the program's structure and is independent from the concrete physical (textual or graphical) representation. Language elements that are used for disambiguation in the concrete syntax are omitted from the AST. As a result, the production rules that describe the structure of an AST, referred to as abstract production rules, are typically simpler than the production rules for the concrete syntax of a given language.

It is typically the AST, not the linear program code, which is the subject of program analysis and transformation, and which is used for code generation by the compiler. Often, formal semantics of a language is defined against the elements of the abstract syntax tree, as it excludes elements of the input language which have no effect on the semantics (parentheses, identation, etc. that are defined in the concrete syntax).

3.1.2 Formal semantics

Formal semantics of a language aims to formally specify the rigorous mathematical meaning of syntactically well-formed sentences in the given programming language. Formal semantics is specified in terms of well understood mathematical concepts, often (but not necessarily) by describing the behavior of a concrete or abstract machine while executing a program in the language. The following section describes several approaches to specifying the formal semantics of programming languages, including translational, operational, denotational, axiomatic, algebraic and action semantics.

The most widely applied approaches to specifying the formal semantics of a programming language are operational, denotational and axiomatic semantics. These approaches and their variants are described in the following paragraphs, based on [73]. For further details and discussion of approaches not described here, please refer to [73].

3.1.3 Operational semantics

Translational semantics

Compilers that convert source code in a given programming language to machine code implicitly specify the semantics of the language. This is, in fact, a translation from a high-level language to a low-level, machine-oriented one, which is closely related to a specific machine architecture. This machine does not need to be an actual, physical machine; an abstract machine with a small number of well-defined primitive constructs will suffice, assuming that they are capable of unambiguously describing the machine's behavior.

One apparent disadvantage of the translational approach is that the semantics of the source language is defined only as well as the target language of the translator. If certain aspects of the semantics of the target language are not clearly understood, semantics of source language constructs that map to these aspects cannot be specified, either. Furthermore, low-level machine code may provide little insight into the essential nature of the source language, as it might not be the proper level of abstraction at which certain properties of a high-level language can be conveniently examined.

Traditional operational semantics

While translational semantics specify what a program does in terms of low-level machine instructions, operational semantics concentrates on how a computation is performed. Operational semantics describe computation using a precisely defined abstract machine, which is specified in terms of mathematical or logical concepts. This abstract machine eliminates the shortcomings of a concrete computer such as limitations on the available memory and storage space, word size, precision of arithmetics, etc., while focuses exclusively how the abstract state of the machine is altered as the program executes.

The basic components of a operational semantics specification are the following:

- an **abstract machine**,
- the **state** (also called the **configuration**) of the machine,
- a dedicated configuration called the **initial state**,
- a **function** that maps one configuration to another,
- and a **final configuration**.

The program the meaning of which is being investigated is represented as a function that iteratively alters the machine state. The final state carries the output of the program.

Structural operational semantics

While traditional operational semantics describes computation in terms of steps of a hypothetical abstract machine, structural operational semantics [65] describes computation by a set of logical deduction rules that turn programs into a set of logical inferences. This allows for proving properties of the language directly from the logical definition of language constructs using logical deduction.

In the structural operational semantics approach, language constructs are described as inference rules: a set of premises, an optional condition and a conclusion. An inference rule with an empty set of premises is called an axiom. Inference rules are used to describe the structure of language constructs similarly to production rules of the grammar that define the syntax of the language. To describe the evaluation of expressions, an abstraction of the memory of a computer, called the *store*, is used. The store is represented as a finite list of numerals. Since the evaluation of an expression does not change the state of the machine, inference rules about the evaluation of expressions do not include the state of the store in the conclusion (the store is read, but not written). Commands that represent a steps of the machine, however, do alter the machine's configuration. Therefore, inference rules describing commands include the current input list, the current output list and the store.

Structural operational semantics allows for reasoning about the semantic equivalence of two language constructs. Semantic equivalence of two constructs holds whenever, for the same initial state, both constructs will drive the abstract machine to the same final state or both will cause the machine to halt. Proving semantic equivalence relies on natural deduction, building up the proof from axioms and inference rules that describe even the smallest details of the changes in the machine's configuration.

3.1.4 Denotational semantics

Based on the observation that both programs and the objects they manipulate are abstract mathematical objects, denotational semantics [74] takes the approach of associating a phrase of the programming language with the mathematical object to which it corresponds (a number, a tuple, a function, etc.). The mathematical object is called the *denotation* of the phrase.

As defined in the abstract production rules of the language, the abstract syntax tree of a phrase corresponding to a language construct consists of subphrases. This hierarchical structure of the language is essential to the specification of denotational semantics: The denotation of a language construct is defined in terms of the denotation of its subphrases.

The specification of the denotation of a phrase can be thought of as a recursive, higher-order function. Prior research on lambda calculus studied higher-order functions extensively, and thus the notations used in denotational semantics borrow much from those in lambda calculus.

Formally, denotational semantics of a language is defined as a mapping between syntactic elements and a semantic domain. For simplicity, syntactic elements are given in terms of concepts in the abstract syntax, not the concrete syntax: The abstract production rules that describe the structure of the AST are simpler and easier to handle than the BNF specification of the grammar. The syntactic categories typically used are, for example, numerals, expressions, commands and identifiers. Each element in the syntactic domain is associated with one of these categories. Abstract production rules describe the possible structure of the elements of the syntactic domain. The semantic domain is defined as sets of mathematical objects, such as boolean or integer values, and functions with precisely defined domains and codomains.

The connection between the syntactic and semantic domains is defined in terms of semantic functions and semantic equations. Semantic functions map objects of the syntactic domain to objects in the semantic domain, while semantic equations describe, using mathematical operations, how the semantic functions behave on different patterns of syntactic objects. For every abstract production rule, a semantic equation defines the meaning of the phrase that corresponds to the production rule. The meaning of a phrase is defined in terms of the meaning of its immediate subphrases. As a result, the denotational semantic specification of a programming language will have a similar structure to that of the abstract production rules of the syntactic elements.

Denotational semantics is a powerful and expressive approach that allows for proving properties of programming languages and the correctness of programs. For example, to prove semantic equivalence of language constructs, it is sufficient to show that they have identical denotations in the semantic domain.

Action semantics

Denotational semantics, as well as many of the previously described approaches, produce notationally dense and sometimes cryptic specifications. Programmers who want to learn or implement a programming language rarely consult its formal semantic specification, since there is a disconnect between the concepts these formal approaches employ and the way programmers view programming languages. In fact, sometimes the most fundamental elements of a language are the hardest to formally describe, such as control flow, continuations, parameter passing or scoping. Formal description of these concepts may become so obscured such that it requires considerable effort to identify them in the specification of semantics. Action semantics [61] was created to tackle this issue. Action semantics is, in fact, a denotational approach, where the constituents of semantic domain directly reflect familiar computational concepts, specifically actions, data and yielders (not yet evaluated pieces of data).

3.1.5 Axiomatic and algebraic semantics

Unlike the previously described approaches, where semantics of a program is described in terms of a real or abstract machine, axiomatic and algebraic approaches to formal specification of semantics aim to specify the meaning of a phrase with predicate logic, without relying on the concept machine state.

Axiomatic semantics

In axiomatic semantics [45], the semantics of a phrase (or a program) is described with logical assertions on values and variables, omitting details on how the computation is carried out. The phrase the semantic meaning of which is being described is tagged with an initial assertion and a final assertion: logical formulas that must evaluate to true before and after the phrase, respectively. The relation between the initial and the final assertions capture the semantics of the phrase.

In axiomatic semantics, semantic equivalence of two phrases does not necessarily require that assuming an identical initial state, the execution of both phrases result in identical final states (or non-termination). Instead, two phrases are equivalent, if the two phrases produce the same final assertions given the initial assertions are the same. The assumptions might not include all variables in the code, and might not require the variables to hold a certain value, as it is merely the invariant relationship between the initial and final assertions that specifies the semantics of a piece of code.

In contrast with operational and denotational semantics, proofs in the axiomatic approach are *static*. That is, while the previous approaches check the program by concentrating on how the state of the machine evolves as the program is being executed, proofs in axiomatic semantics can be elaborated by static analysis of the source code of the program.

Beside semantic equivalence and correctness proofs, the axiomatic semantics approach provides a means to formal specification of programs. Instead of describing the semantic meaning of programs that already exists, this technique can be used in the reverse situation: given the initial and final assertions, derive a program for which these assertions hold.

Algebraic semantics

Algebraic semantics [38] takes a similar approach to that of axiomatic semantics in the sense that the specification of programs is expressed without relying on the concept of a machine. However, while axiomatic semantics builds on predicate logic, the theoretical foundations of algebraic semantics lie in abstract algebra. Instead of using only logical assertions over values and variables, algebraic semantics rely on describing the properties of operations over abstract objects.

The algebraic approach is naturally applicable to simple, low-level objects such as Boolean values with operations on them such as conjunction, disjunction, implication, etc., but lends itself to easily describing more complex operations on abstract data types (ADT) , as well. In fact, algebraic semantics is an ideal vehicle for the specification of ADTs, because the specification omits specifics of the actual representation of data and the implementation of the operations, while focuses on the properties of operations that manipulate data. This aligns well with the objectives of the object-oriented programming paradigm, promoting information hiding through encapsulation and polymorphism.

3.2 Problem statement

The previous chapter described the main ideas behind the design of TinyVT, along with source code examples and the description of the behavior of the machine executing these pieces of code. However, the fact that this description is given informally in a natural language, renders it inadequate as a formal semantics specification. While this informal description is a good starting point to learn and understand TinyVT, and is even helpful when implementing a TinyVT compiler, by no means is it guaranteed to be free from ambiguities or from the overspecification of language features.

Since TinyVT is an extension of the C language, the semantics of which has already been specified, TinyVT's formal semantics can be given by building on an existing formal semantics specification for C.

The notion of a thread, defined as an independent unit of computation with conceptually linear control flow, is missing from C, since the C language is a legacy of an era in which multithreading had not yet existed. Today, thread support in C is provided in the form of external libraries. Specification of semantics of such systems, however, proved to be problematic. In fact, Boehm argues that, for languages that were originally designed without thread support and to which a library of threading primitives was later added, a pure, library-based threading approach, in general, cannot guarantee correctness of the resulting code [9].

TinyVT, however, takes a nontraditional approach to providing the thread abstraction.

TinyVT provides language constructs that allow for describing computation by defining the control flow using C control structures, assuming that the computation has an independent execution thread. Unlike traditional multithreading, where the abstraction of a virtual processor is provided by the operating system or a user-space threading library, the abstraction of the local execution thread that TinyVT offers is provided by the language and the compiler. The TinyVT compiler is a source-to-source translator, which translates the source code relying on the thread abstraction to plain C code, by rewriting TinyVT thread definitions as a set of event handlers. Since threads are resolved to C code that is assumed to be run in a single-threaded manner, Boehm's observation does not apply to TinyVT. TinyVT's semantics specification can, therefore, allow for investigating threading-related properties of systems, such as interleaving of thread execution and interaction between threads.

The specification of TinyVT's formal semantics includes the following four areas.

- **Semantics of TinyVT-specific constructs.** TinyVT extends the C language with a number of new language constructs that are used to define threads, to communicate between a thread and its environment (which may include other threads, software entities external to a thread, or hardware), and to manage control flow. It is crucial that the semantics of these language constructs be precisely specified.

- **Semantics of ANSI C constructs.** Most but not all C language constructs are allowed within TinyVT threads. A subset of those that are allowed, however, have different semantics when used within a TinyVT thread than the original C semantics. Although TinyVT's specification of semantics can reuse elements from a formal semantics specification for C, it is essential that all such differences be unambiguously described.

- **Interaction semantics of TinyVT threads.** TinyVT extends the C language with threading support, however, the concept of threads is not present in the C language. Semantics of interaction between a thread and its environment needs to be formally described, with particular interest in control flow and communication semantics.

- **Compositional semantics of TinyVT threads.** TinyVT threads can be composed to form a composite software module. We are interested in the semantics of composition, particularly in identifying properties that are preserved through composition of TinyVT threads. Furthermore, the compositional semantics should allow for investigating the factors that influence whether a composition of a set of threads or a complete system (which also includes the event-driven runtime) is deterministic or not (with a suitable definition of determinism).

3.3 Organization

This section is organized as follows. First, the approach to specifying the formal semantics of TintVT is outlined. I argue that the operational approach with a specification that is based on abstract state machines (ASM) can meet the requirements described in the problem statement. Following a brief introduction to abstract state machines and the AsmL language, a formal specification of the C language, given by Gurevich and Huggins, is described. I give the formal semantics of TinyVT by extending this specification to include the semantics of the language extensions introduced by TinyVT.

Compositionality of TinyVT threads is explored at a higher level of abstraction than the abstraction levels used in the specification of semantics of C by Gurevich and Huggins. Therefore, following the work of Chen et al. on semantic anchoring [14], I describe a mapping from TinyVT threads to a finite automata based model, the behavioral and compositional semantics of which I define formally in the AsmL language. This approach allows for examining the properties of composition of threads as parallel composition of finite automata.

Finally, I will show that the finite automaton to which a TinyVT thread is mapped is always deterministic (i.e. for the same initial configuration, whenever two traces agree on the inputs they will agree on the outputs and the final state as well), and that composition preserves determinism.

3.4 Approach

3.4.1 Alternatives

One of the most obvious means of specifying the formal semantics of a programming language is the translational approach. This can be achieved by choosing a target language for which a formal specification of semantics already exists. Then, one needs to formally describe a set of translation rules that map TinyVT program code to the target language. This approach would certainly be suitable to specify TinyVT's semantics: C lends itself to being the target language of the translation; and the formal description of the TinyVT compiler could serve as a set of transformation rules. While such a formal semantics specification adequately describes a particular compiler, it tends to be too inflexible as a language specification. This approach implies that all compilers must implement certain language features in one particular way, resulting in unnecessarily overspecifying the language. Another disadvantage of the translational approach is that the resulting specification is very cumbersome to reason about: it would not be the right level of abstraction to examine the interaction semantics and compositional semantics of TinyVT threads.

The operational and denotational approaches eliminate these shortcomings of the translational approach, since they are used to specify the semantics of a language formally, in terms of logical or mathematical concepts. Significant research has been conducted on formally specifying the semantics of the C language, using the operational and the denotational approaches: Gurevich and Huggins gave the formal (operational) semantics of C using the abstract state machines approach (formerly called evolving algebras) [35]. Norrish formalized the operational semantics of C in Isabelle/HOL [63]. Sethi used the denotational approach to describe the semantics of C control structures and declarations [70]. Cook and Subramanian formalized the semantics of a subset of C in the Boyer-Moore theorem prover [18]. Cook et al. used the denotational approach to derive a denotational semantic specification for C in temporal logic [17]. Similarly, Papaspyrou, in his Ph.D. thesis [64], provides a complete denotational semantics specification or C.

3.4.2 Operational semantics with abstract state machines

To specify the formal semantics of TinyVT, I chose to extend the work of Gurevich and Huggins, which follows the operational approach. This work specifies the semantics of C using abstract state machines (evolving algebras). One compelling property of an abstract state machines based specification is that the semantics can be described on multiple abstraction levels, where a lower abstraction level is a refinement of a higher one. In their work, Gurevich and Huggins used four abstraction layers, which specify the semantics of control flow, evaluation of expression, memory allocation and initialization, and function invocations, respectively. Although these four layers are sufficient to describe the formal semantics of TinyVT, exploring the interaction and compositional semantics of threads requires a higher level of abstraction, where threads are handled as first class objects while omitting unnecessary details.

In this work, I specify the semantics of TinyVT using five abstraction layers. Within the the same framework that Gurevich and Huggins used, I describe the semantics of control flow, evaluation of expression, memory allocation and initialization, and function invocations in TinyVT. To investigate the interaction semantics and compositional semantics of TinyVT threads, I introduce a fifth layer of abstraction.

3.4.3 Modeling threads as automata

TinyVT threads are software artifacts with a reactive behavior: The execution of a thread is triggered (or resumed) by an event from the thread's environment. As a reaction to this event, the thread carries out a computation, alters the local state, and returns control to the originator of the event. While carrying out a computation, a thread may send events to its environment (which may include other threads, other software or hardware entities). The thread's reaction to an external event depends on the local state. The state of a thread is not exposed to its environment: local state can only be altered by the thread's local computation.

The above characteristics of TinyVT threads make them an ideal subject of being modeled as finite automata (FA): Events to which the thread reacts are modeled as input actions, while events that the thread generates are modeled as output actions. I model returns from threads (after completing a computation in reaction to an external event) as output actions, while returns from the environment (in response to an event from the thread) are modeled as input actions. The automaton has a local state which can only be altered in response to an input, and only in ways defined by the automaton's transition relation. I model compositions of TinyVT threads as parallel composition of finite automata. When two threads are composed, outputs of one may be inputs to the other. One important property of the definition of automata composition I define is that, in such a case, the corresponding input and output actions are instantaneous.

Structurally, I define the FA that I use on the fifth layer of TinyVT's specification of semantics as a 6-tuple of state set, initial state, input, output and internal actions, transition relation. I specify this static structure (also called static semantics or structural semantics), as well as the corresponding behavior (dynamic semantics or behavioral semantics) in the AsmL language.

Given an automaton as a (concrete) set of states, initial state, actions and transitions, the behavioral semantics unambiguously specifies how it operates. Due to the separation of structural and behavioral semantic specifications, however, the behavioral semantics specification does not use the concrete data model of the automaton, since it is specified only in terms of concepts specified in the structural semantics. As a result of this, to assign formal behavioral semantics to TinyVT threads, it is sufficient to describe a mapping from a thread (or more precisely, from the abstract syntax tree of a thread) to the structural model of the finite automata.

3.4.4 Compositionality

Similarly, I define the structural and behavioral semantics of automata composition separately. The static structure of the composition of two finite automata is defined uniquely by the static structure of its parts, where actions are matched by name. When specifying the structural semantics of the composite in AsmL, I do not compute the state set and transition relation of the composite explicitly. This computation can be omitted because the result is never used in the specification of behavioral semantics of composition. The behavioral specification defines how a composite reacts to external events, specifically, how external events are dispatched to the parts and how events that are shared between parts are handled. The state set and transitions of the composite are implicitly defined by the behavioral semantics. Assigning behavioral semantics to a composition of TinyVT threads, it is sufficient to provide a mapping from a set of interacting threads to the data models of a set of corresponding finite automata.

I show that the composition of finite automata is also a finite automaton. This allows for hierarchically modeling composition of TinyVT threads, that is, not only automata, but also compositions of automata can be parts of a composition. I will also show that the mapping from TinyVT thread to automaton always results in a deterministic finite automaton, and that determinism is preserved through composition. As a result, a system that consists exclusively of TinyVT threads is always deterministic.

3.5 Abstract State Machines

Abstract State Machines (formerly known as evolving algebras) [34, 10] is a mathematical formalism which allows for describing arbitrary states of arbitrary algorithms on their natural abstraction level. Abstract state machines (ASM) allow for separating concerns that are at different abstraction levels, e.g. specification-level concerns from design-level concerns, without introducing a gap between the different levels. Data and operations can be represented in terms of the concepts of the particular problem domain in an abstract manner, that is, there is no prescribed means of representing objects and actions.

3.5.1 Mathematical background

State

The notion of state in ASM terminology is not an indivisible entity, but rather an arbitrarily complex (or simple) first-order structure. ASM state is significantly more general than a state of a finite state machine (FSM), or than being just a set or a function. A state is a collection of domains (sets, called universes in ASM terminology, each of which represents a particular kind of object) along with relations and functions defined on them. The number of the universes together with their integrity constraints, and the functions with their arity, domain, and range, are considered as part of the signature of the state. A universe can be completely abstract, meaning that there is no knowledge available about the elements and about their representation in a certain language or system. Alternatively, if the domain elements have certain properties, or are in relation with other objects, or are subject to manipulation, then the corresponding constraints, predicates or functions need to be formalized. The ASM approach, however, does not designate any particular notation for this formalization. Functions are either static, which means that the value of the function can not change as the state evolves, or dynamic, meaning that function values can be altered. Functions with Boolean values are called predicates, which can be used to represent various constraints. Boolean operations, the equality sign, and static names true, false and undef are always part of the vocabulary. The universes and the static functions provide the basic structure of the modeled system, while dynamic functions, which change as the system evolves, reflect the system's dynamic aspect.

Updates

An abstract machine operates by changing the abstract state, that is, by changing the structures. Through these changes, the signature of the state, as well as the predicates (which can be treated as characteristic functions) must remain fixed. It is the functions that can be altered, namely the value of certain functions for certain arguments can be changed. Notice that changing the value of a variable is just a specialization of this concept: The variable can be represented as a nullary function, changing the value of which alters the value of the variable.

State transformation is achieved by the simultaneous execution of finitely many rules. According to Gurevich's definition, ASM M is a finite set of rules that define guarded function updates. Applying one step to state $\mathcal{A}$ produces the next state $\mathcal{A}'$ of the same signature, as follows. First, all guards of the rules of M are evaluated in $\mathcal{A}$, according to the standard interpretation of classical logic. Then, for all rules for which the guard evaluated true, all arguments and update values are computed in $\mathcal{A}$. Finally, the function values in $\mathcal{A}$ are replaced simultaneously by the newly computed update values for the arguments in question (assuming no contradicting updates), yielding $\mathcal{A}'$. As a result of a step, $\mathcal{A}'$ will differ from $\mathcal{A}$ in the values of the functions updated by a rule in M that could fire in $\mathcal{A}$.

Simultaneous execution of rules

The fact that updates within a step are executed simultaneously proved to be particularly useful in many application areas of ASMs. It allows for modeling updates as macrosteps: At a particular level of abstraction, one intends to hide the low-level implementation details (microsteps), which results in simpler high-level models omitting unnecessary details and premature sequentialization. Also, it provides a natural way to model synchronous systems, where a global clock tick can be modeled as a single machine step.

Locality of updates

As a consequence of ASM state not being an indivisible mathematical entity, executing an update step only changes a part of the state, i.e. only a few functions for selected arguments, that appear in the rules the guard predicates of which evaluated true in the given step. Everything not affected by the rules remains unchanged. This, with careful design of models, can prevent combinatorial explosion of state space, which is a common problem with many formal modeling languages that rely on a global, holistic interpretation of systems. This feature of the ASM approach promote modularizing ASM models, such that most updates local to a module will change variables that are only used locally.

Nondeterminism

Nondeterminism is a notion that is essential to modeling reactive systems, or, in general, systems at a high-level of abstraction without prematurely committing to certain design decisions. To support nondeterminism, ASM provides the *choose* rule, with multiple subrules, exactly one of which will be chosen nondeterministically to be executed.

3.5.2 The Abstract State Machine Language

Although the ASM approach does not specify what formalism should be used to describe ASM models, it is convenient to use one of the ASM-based tools (ASM WorkBench [12], XASM [4], ASMGofer [69], AsmL [37]). The Abstract State Machine language (AsmL), developed at Microsoft Research, is an executable specification language based on the theory of Abstract State Machines. Below a brief overview of AsmL is presented, only to the extent required for understanding AsmL code later in this chapter. Detailed description of the language is beyond the scope of this work. For in-depth details please refer to [36].

The syntax of AsmL resembles that of imperative, object-oriented programming languages. It borrows many features from modern object-oriented languages, such as interfaces, classes, inheritance, overloaded functions and operators, etc. AsmL also supports properties (as in C#), exception handling and assertions.

AsmL defines basic types - such as `Boolean`, `Integer` or `String` - and allows for the definition of user defined types. Operations on built-in types are available either natively in the AsmL language, or through the AsmL library.

Types

Universes in ASM models are represented as types in AsmL. The language has built-in types, such as `Null`, `Integer`, `String`, etc. By default, a variable of type `T` cannot have an undefined value (`null`). To allow a variable to hold `null` value, the type should be specified with the "?" type modifier, as "T?".In addition, it provides three type families for collections of values: Sets, sequences and maps can be defined as follows.

- `Set of T` - Unordered, finite collections of distinct elements of type `T`
- `Seq of T` - Ordered, finite sequences of elements of type `T`
- `Map of T to S` - Tables that map distinct keys of type `T` to values of type `S`

There are several ways to create user-defined types. Tuples can be defined in the form of `(T1,T2)` where both `T1` and `T2` are types. Alternatively, arbitrary user-defined compound types can be defined using the structure keyword:

```
structure Person
 name as String
 age as Integer
```

The third alternative to create user-defined types is the notion of class. Classes in AsmL can be defined similarly to classes in other object-oriented languages. A class definition contains both data (fields) and operations on the date (methods).

```
class Vector2D
 var X as Integer
 var Y as Integer
```

AsmL supports inheritance (but not multiple inheritance):

```
class Vector3D extends Vector2D
 var Z as Integer
```

For all types except for classes, the semantics of equality is based on value. Two variables are equal if they have the same structure and the values of the elements are equal. In contrast, two instances of a class are never equal. Classes have reference semantics. There are no pointers in AsmL, hence, classes provide the only means to share memory, and it is the only form of aliasing available.

One particularly helpful language feature is that classes can be defined incrementally. For example, the two code segments below are equivalent.

```
class Circle
 var O as Integer
 var R as Integer

class Circle
 var isFilled as Boolean
```

```
class Circle
 var O as Integer
 var R as Integer
 var isFilled as Boolean
```

In addition to incremental additions to a class definition, incremental modifications (e.g. adding modifiers or adding an interface the class implements) are also allowed.

Variables

Variables in AsmL are equivalent to dynamic nullary functions in the underlying ASM model. To declare a variable of a simple type, one would write, for example:

```
var i as Integer
```

Depending on the scope, variables can be global, local or instance-based. Global variables are accessible from all code, while local variables are only accessible from within the block where they are defined. Instance-based variables are accessible through their encapsulating object using the '.' operator.

Updates

Execution of AsmL programs progresses in discrete steps. Updates do not occur until the step in which they are executed is completed, therefore, the updated value of a variable can only be observed in the next step. The following piece of AsmL code demonstrates this.

```
var a as Integer = 0
var b as Integer = 0

Main()
 step
  a := 1
  b := 2
  WriteLine(a)  // a is still 0 here
  WriteLine(b)  // b is still 0 here
 step
  // updates of previous step visible here
  WriteLine(a)  // a is 1 here
  WriteLine(b)  // b is 2 here
 step
  // swap values of a and b
  a := b
  b := a
 step
  // updates of previous step visible here
  WriteLine(a)  // a is 2 here
  WriteLine(b)  // b is 1 here
```

This synchronous deferred update semantics allows for swapping the values of two variables without using a temporary variable. Consider the fourth step in the above example: Since updates only occur at the end of the step, the values of `a` and `b` on the left-hand-side of the update operation will evaluate to `1` and `2`. The update of the variables will be deferred until the end of the step.

Reflecting ASM's approach to function updates, all variable updates that are executed within a single step are simultaneous in AsmL. Updates in AsmL can be either complete or partial. Multiple partial updates are allowed within an execution step as long as they are consistent.

The following example demonstrates a complete update of a variable of a structured type.

```
var p as Person = Person("Jane Doe", 33)

Main()
 step
  p := Person("Jane Smith", 34)
```

Alternatively, the code below, containing two consistent partial updates has the same effect.

```
var p as Person = Person("Jane Doe", 33)

Main()
 step
  p.name := "Jane Smith"
  p.age := 34
```

Methods

Methods are named operations that can be invoked in various contexts. A method definition includes the name of the method, and may optionally specify a finite number of arguments and a return value. AsmL distinguishes two kinds of operations: functions and update procedures. Although syntactically equivalent, functions have no effect on the state variables, while execution of update procedures alters the values of state variables after the update step is completed. An update procedure that increments the value of global variable `i` with a delta value given as a parameter is programmed as follows.

```
Increment(delta as Integer)
 i := i + delta
```

A dedicated function, Main() serves as a global entry point to an AsmL program.

3.6 Operational semantics of C

In [35] Gurevich and Huggins specify the formal operational semantics of C using the Abstract State Machine approach[1]. An ASM based formal specification may include several layers of abstraction, each layer being the refinement of a higher-level one. This way, language features can be examined at the desired level of abstraction at which irrelevant details are omitted. This layering, at the same time, gives better structure to the formal specification, and makes it easier to comprehend.

The specification of C semantics by Gurevich and Huggins is only concerned with behavioral aspects of a C program, and assumes that all syntactic information is resolved by the syntactic analyser, and is available to the ASMs that define the program behavior. Instead of operating on an abstract syntax tree (AST) of the C code, the ASMs assume that the

[1][35] uses the term Evolving Algebras, since it was written before the approach was renamed to Abstract State Machines. For the sake of clarity, I use the term Abstract State Machine is used in this dissertation.

syntactic analyser outputs the static data structures (a set of static functions) on which the ASMs operate.

The C semantics specified by Gurevich and Huggins is comprised of four layers:

- Statements
- Expressions
- Memory allocation and initialization
- Functions

The rest of this section gives a brief overview of each of these layers, highlighting the techniques the authors used, and focusing on the details which are required to understand TinyVT's specification of semantics, which will be presented later in this chapter.

3.6.1 Layer 1: Statements

The first layer models C statements, including those that define C control structures (`do`, `while`, `for`, `if`, etc.). Two universes are defined at this layer of abstraction: *tasks* and *tags*. Tasks represent units of computation by the C program interpreter, such as execution of a C statement, evaluation of an expression or initialization of a variable. The set of tasks contains all tasks that may occur during the execution of the program, and it depends on a particular program being executed. A distinguished dynamic nullary function, $CurTask$: *task* indicates the current task. The static function $NextTask$: *task* $\to$ *task* ensures that tasks are executed in the given order. (The abstract syntax of a given C program unambiguously defines the $NextTask$ function.) Initially, at this layer of abstraction, $CurTask$ is set to the first statement in the program. After the last statement of the program, $CurTask$ is set to $undef$.

Transferring control to a specific task by modifying the value of $CurTask$ is a recurrent theme in this specification. Gurevich and Huggins define the `MoveTo` macro that transfers control to the task given as parameter:

```
MoveTo(Task)
 CurTask := Task
```

The universe tags contains labels that are assigned to tasks with the static $TaskType$: *task* $\to$ *tag* function, indicating the nature of the task (e.g. wether the task is an execution of a statement or an initialization of a variable, etc.).

This layer of abstraction specifies the semantics of all types of C statements: expression, selection, iteration, jump, labeled and compound statements. The following paragraphs explain, through a representative subset of the above statement kinds, how the ASM approach is used to specify the semantics of C. For further details, the reader should refer to [35].

Expression statements

In C, an expression statement means evaluating an expression. At this level of abstraction, it is assumed that the evaluation of the expression is handled by an external function $TestValue : task \rightarrow result$, where result is an universe of *results*. The expression is evaluated even if the resulting value is never used, since the evaluation may have side-effects. As this layer of abstraction of the semantics specification is only concerned with control flow, the ASM rule for an expression statement is as simple as proceeding to the next task.

```
if TaskType(CurTask) = expression then
 MoveTo(NextTask(CurTask))
```

Selection statements

C specifies two kinds of selection statements: if and switch. The if statement has two forms: "`if(expression) statement`" and "`if(expression) statement else statement`". To give and idea how the semantics of selection statements is defined, without being complete, I only present the semantics of the latter here.

The semantics of the if-else statement relies on the external function $TestValue$ to evaluate the guard expression. If the result TestValue returns is non-zero, the task in the true branch is executed. Otherwise, if the result is zero, execution continues with the false branch. The static functions $TrueTask : task \rightarrow task$ and $FalseTask : task \rightarrow task$ are used to query for the true and false branches in an if statement.

```
if TaskType(CurTask) = branch then
 if TestValue(CurTask) != 0 then
  MoveTo(TrueTask(CurTask))
 elseif TestValue(CurTask) = 0 then
  MoveTo(FalseTask(CurTask))
```

The statements in both branches link to the task following the if statement with the static $NextTask$ function.

Statements in the true or false branches can potentially be compound statements. Gurevich and Huggins do not give special rules for compound statements: A compound statement is a list of statements linked together with the static $NextTask$ function. The first task within the compound statement is linked from the last task preceding it, and the last task of the compound statement is linked to the first task following it.

Iteration statements

Similar to the if statement, the semantics of iteration statements are also specified using the $TestValue$, $TrueTask$ and $FalseTask$ functions.

Jump statements

The `break` statement within a `switch` statement or a `goto` statement indicate that control should be unconditionally transferred to the task specified by the default label, or by the corresponding label, respectively. The `break` and `continue` statements unconditionally transfer control to the first task of the enclosing iteration statement or the first task following the enclosing iteration statement, respectively. All this information is available statically, and is encoded in the static $NextTask$ function.

The return statement, at this layer of abstraction, is modeled by setting $CurTask$ to $undef$ and halting program execution.

3.6.2 Layer 2: Expressions

The second layer of abstraction in the operational semantics specification of C deals with expressions. Refining the first layer, the $TestValue$ function is concretized: At the second layer, it is an internal dynamic function. Furthermore, tasks with expression tags (which model expression statements in the first layer) are now expanded, representing the internal structure of expressions. This level of abstraction incorporates the notion of a store abstraction, and uses several functions to represent memory read/write operations. Also, C built-in types are handled at this level, with static functions to return the size of a type and to convert memory locations to results of a given type. Identifiers (identifier expressions) are mapped to memory locations, using a static function, as well.

All kinds of expressions are modeled at the second layer, except for function invocations. For now, a function invocation is modeled with an external $FunctionValue : task \rightarrow result$ function, which returns the result, i.e. the return value of the function.

Evaluation order of subexpressions

Undefined evaluation order

According to the C standard, for many binary expressions in C (binary arithmetic operations, assignment operation, etc.) the order of evaluation of subexpressions is undefined. This means that C compilers are free to generate code with arbitrary fixed evaluation order, or, can take advantage of this ambiguity to implement compiler optimizations. As a result, for the same source code, the evaluation order of subexpressions may be different from platform to platform (hardware and operating system), moreover, it may even vary between subsequent executions of the same binary on the same platform.

The ASM based semantics specification uses the `choose` construct to model this nondeterminism. For all binary operations with undefined execution order, a dynamic function $Visited : task \rightarrow \{neither, left, right, both\}$ is used to mark which subexpressions have already been evaluated. Initially, $Visited$ is set to *neither*, and a nondeterministic choice is made to decide if the left or the right subexpression is to evaluate first. After evaluating the chosen subexpression, the value of the $Visited$ function is updated for the parent statement accordingly. Then, the other subexpression is evaluated, setting the value of $Visited$ to *both*. To allow for jumping between subexpressions depending on the nondeterministic choice, the $MoveTo$ macro is redefined, such that it inspect the value of $Visited$ to set $CurTask$ to the subexpression which has not been evaluated yet.

Omitted subexpressions

For some expressions, subexpressions may or may not be evaluated, depending on certain conditions. For example, if the first (left) operand of a logical `OR` expression evaluates to a non-zero value (`TRUE`), the result of the expression is known, hence the second (right) operand will not be evaluated. Similarly, for the logical `AND` operation, the evaluation of the second operand is omitted if the first operand evaluates to zero (`FALSE`). Gurevich and Huggins model this by linking the subexpressions with $TrueTask$ and $FalseTask$ instead of $NextTask$, hence, skipping the evaluation of the left operand based on the result from the evaluation of the right operand.

3.6.3 Layer 3: Memory allocation and initialization

The third layer extends the previous layer with the semantics of memory allocation and initialization. The *tags* universe is extended with the *declaration* element, representing variable declaration tasks. Declaration tasks are linked in the proper order with statement tasks with the static $NextTask$ function.

C distinguishes between static and non-static variables. Static variables are initialized at most once, only the first time the declaration task is executed. Every time the declaration task of a static variable is executed, it assigns the same memory area to the declared variable. For non-static variables, the declaration task executes the initializer unconditionally, and assigns a new memory area to the variable.

Special care is needed when treating local automatic variables that are declared (with optional initializers) in a scope that can be entered with a non-local jump. In such case, memory must be allocated to the variable, but the initialization, if present, is skipped. Gurevich and Huggins solve this by redefining the semantics of the non-local goto statement: Instead of an unconditional jump, the goto statement now transfers control to a series of indirect initialization task that allocate the memory for the local automatic variables within the scope being entered.

3.6.4 Layer 4: Functions

The fourth layer of abstraction specifies the semantics of function definitions and function invocations. Since a C function may have multiple active incarnations at a given moment (e.g. as a result of recursion), a task alone is insufficient to capture to which incarnation of the function it belongs. Overcoming this issue requires modeling the stack as a universe. The universe *stack* consists of positive integers, each representing a different stack frame, whith a distinguished element $StackRoot = 1$. The functions $StackNext : stack \rightarrow stack$ and $StackPrev : stack \rightarrow stack$ are used to navigate the stack. The unary function $StackTop$ represents the top of the stack. Store-related functions are now changed to reflect state stored on the stack. The $FunctionValue$ function, introduced in the second layer of abstraction is now eliminated.

Function invocation, from the caller's point of view, is modeled as follows. Similarly to most binary operation expressions, the evaluation order of function arguments is undefined in C. This nondeterministic evaluation is modeled analogously to binary operation expressions in the second layer. Once the arguments are computed, a new frame is pushed to the stack by incrementing $StackTop$. The values of the arguments, as well as the return task, are associated with the new top of the stack, and control is transferred to the first task of the function. The $ReturnTask : stack \rightarrow task$ function is updated to return the task following the function invocation expression for the current top of the stack. When the function finishes, control is returned to this task. After the function returns, the result is available at the top of the stack.

From the callee's aspect, execution of a function consists of three steps. First, memory is allocated for the arguments, as described in layer three. Then, the function's body (a compound statement) is executed. Finally, the return statement is redefined to associate the optional return value with the top of the stack, and pass control back to the return task associated with the top of the stack.

Since previous abstraction layers did not handle functions, there was no distinction between local and global variables. This layer of abstraction defines a static $GlobalVar$:

task → *bool* function to check if a variable is global or not. The specification of semantics assigns all global variables to *StackRoot*. This way, when global variables need to be accessed, *StackRoot* is used instead of the actual top of the stack in the corresponding functions. To bootstrap a C program, this abstraction layer of the semantics specification sets the initial value of *CurTask* to the first global variable declaration, or, if none present, to the first task of the *main*() function.

3.7 Semantics of TinyVT

This section defines the semantics of TinyVT as an extension to the C semantics of Gurevich and Huggins. The four levels of abstraction allow for exploring distinct aspects of the semantics of TinyVT separately.

3.7.1 Layer 1: Statements

The first layer of abstraction models C control structures. TinyVT extends the C language with four statements: `await`, `yield`, `dreturn` and `ireturn`.

The await statement

We will model the `await` statement as a selection statement: Since an `await` statement may include multiple event handlers, the type of the received event will specify which one of the inlined function bodies will be executed. Because the semantics of function invocations are not specified until the fourth layer, for now, we define an external *ResumeTask* : *task* → *task* function which, for an await task, returns the first task of the inlined event handler the thread execution should continue with.

```
if TaskType(CurTask) = await then
 Moveto(ResumeTask(CurTask))
```

ResumeTask may return undef in the case when an unexpected event is received by the thread. In such situation, the execution of the program halts.

The yield statement

The `yield` statement is syntactic sugar. It is equivalent to requesting the deferred execution of the *deferredEvent* event handler, followed by an `await` statement with a single *deferredEvent* handler inlined. The deferred execution request is a function call to the event dispatcher (a service external to a thread), which is handled as an expression statement at this abstraction level. This expression statement is linked to the await statement with an empty-bodied event handler of *deferredEvent* with *NextTask*.

The ireturn statement

TinyVT's syntax does not allow the return statement within threads. Instead, two return-like constructs are specified: `ireturn` and `dreturn`.

The `dreturn` statement may appear in the body of inlined event handlers within `await` statements. It may or may not be followed by an expression defining the return value. The `ireturn` statement is a shorthand notation to specify that a yield statement should be executed immediately after the enclosing `await` statement. A more complete discussion of the `ireturn` (and `dreturn`) will be presented in the fourth layer of abstraction. For now, `ireturn` is specified as an unconditional jump to a yield operation, which is then linked with $NextTask$ to the task following the enclosing `await` block.

The dreturn statement

Similarly to the syntax of `ireturn`, the `dreturn` statement may appear in the body of inlined event handlers within `await` statements, and may or may not be followed by an expression defining the return value. It is modeled as an unconditional jump to the task following the enclosing `await` block.

Limitations on jump statements

TinyVT syntax does not allow for jumps into or out of the body of inlined event handlers of **await** statements. This restriction applies to goto, break and continue statements, and forbids switch statements the body of which incudes an await statement with a case label in the inlined event handler.

For other uses of jump statements, the semantics defined by Gurevich and Huggins apply.

3.7.2 Layer 2: Expressions

For many binary operations, the C standard does not define an evaluation order for the subexpressions representing the operands.

TinyVT, however, requires that the order of function calls made by the thread should always be deterministic, therefore, expressions that include function call subexpressions with undefined evaluation order are not allowed in the generated code.

3.7.3 Layer 3: Memory allocation and initialization

Memory allocation and initialization within TinyVT threads is identical to that in standard C.

3.7.4 Layer 4: Functions

The caller's story

Similarly to binary mathematical operations, for function invocations, the C standard does not define a fixed evaluation order of arguments. TinyVT, however, forbids this nondeterminism by requiring that the order of function calls made by the thread be deterministic. Apart from the evaluation order of function arguments, the caller's story is identical that in the specification given by Gurevich and Huggins.

The callee's story

Event handlers in TinyVT are similar to C function definitions in many respect. However, there can be multiple different event handlers for the same event type inlined in different await statement. It depends on the actual thread state which of these is going to be executed in response to a function call from the environment.

In response to a function call from the environment, control is passed to an event handler stub within the thread that is common to all events of the same kind. First, memory is allocated for the parameters, identically to C function definitions. Then, memory is allocated to the return value if the return type is non-void.

We define a dynamic function $\mathit{ThreadState} : \mathit{task}$ which defines where the execution of the thread should be resumed at the next call to the thread. Initially, $\mathit{ThreadState}$ is set to the task corresponding to the first `await` statement of the thread.

A static function $\mathit{ResumeTask} : (\mathit{task}, \mathit{task}) \rightarrow \mathit{task}$ is called to find out which handler body of `which` await block to jump to. The first argument of $\mathit{ResumeTask}$ is the current task that identifies the event handler stub, the second argument is the current thread state, and it returns the first task of a particular event handler within the await block at which the thread is currently blocked.

The semantics of `dreturn` and `ireturn` statements are expanded at this level of abstraction. If the return statement is followed by an expression, it is evaluated, and the result is placed in the memory allocated for the return value. After this, the control leaves the inlined function body as it is specified in the first layer.

The execution of the function ends when an `await` statement is reached. At this point, $\mathit{ThreadState}$ is set to the identifier of the await task reached, the return value is associated with the top of the stack and control is passed to $\mathit{ReturnTask}$.

3.8 Compositionality

Compositionality is an important notion in designing and analyzing complex systems. For a reasonably complex system that is built of a large number of components, proving that certain properties hold for the system as a whole can be very complicated. Compositionality provides a constructive approach to proving system properties. Instead of the entire system being the subject of analysis, it is sufficient to ensure that the properties in question hold for its constituents, commonly called components, if it can be shown that properties are preserved through composition. The essence of compositionality is that properties of the composite are a function of properties of its components.

In this section, the compositionality of TinyVT threads is investigated. Since ensuring predictable operation is one of the fundamental design goals in embedded systems, the property I will closely look at is determinism. I define determinism as follows. A system is deterministic if, whenever two program traces agree on the inputs, they always agree on the outputs and the final state, as well. While such a definition of determinism is meaningless in embedded systems in general, since not only the ordering but the timing of the inputs affect the outputs, it is suitable for TinyVT threads that are shielded from the environment with an event-driven runtime that serializes external events. I will show that show that TinyVT threads are deterministic in this sense, and that the (parallel) composition operation that I define preserves this property.

While the previously presented four abstraction layers of the semantic specification sufficiently describe the semantics of TinyVT, the level of detail is too fine-grained to examine the compositional behavior of TinyVT threads. The ASM approach, however, provides a means to describe the semantics of the language at the level of abstraction that is most suited to investigate the property in question. This section describes TinyVT automata (TA), an abstract model that hides irrelevant details and allows for examining interaction between different threads and between threads and the environment, focusing on control flow and communication. The abstract model retains details that are related to externally observable communication and control, such as tasks related to passing control (i.e. calling functions external to a thread or awaiting external events) and local thread state that affects control flow. Irrelevant details, such as tasks corresponding to individual C statements that are not related to passing control across thread boundaries, or local state not affecting control flow are not modeled.

The model described below allows for hierarchical composition, that is, a composition of threads may also be subject to composition. This way, a complex system can be modeled as a hierarchical composition where the leaves of the composition tree are TinyVT threads and the non-leaf nodes nodes are composites.

3.8.1 Modeling TinyVT threads as finite automata

Formally, a TinyVT thread is modeled as a TinyVT Automaton (TA), a kind of finite automaton similar to I/O Automaton [57] and Interface Automaton [19], but with different behavioral semantics. The TA is defined as a 6-tuple $< S, s_0, A_{in}, A_{out}, A_h, T >$, where

- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- A_{in} is a finite set of input actions, A_{out} is a finite set of output actions, and A_h is a finite set of internal (hidden) actions. The set of input, output and internal actions are pairwise disjoint, that is, $A_{in} \cap A_{out} = \oslash$, $A_{in} \cap A_h = \oslash$ and $A_{out} \cap A_h = \oslash$. $A = A_{in} \cup A_{out} \cup A_h$ denotes the set of all actions.
- $T : S \times A \times S$ is a transition relation.

If $a \in A_{in}$, then $(s_i, a, s_j) \in T$ is called an *input transition*. Similarly, if $a \in A_{out}$ or $a \in A_h$, then $(s_i, a, s_j) \in T$ is called an *output* or *internal transition*, respectively. An action a is *enabled* in state s if there exists a transition (s_i, a, s_j) for some state s_j.

We denote the set of enabled input, output and internal actions in state s with $A_{in}(s)$, $A_{out}(s)$ and $A_h(s)$, respectively. The set of enabled actions at state s is denoted with $A(s)$.

Similarly to Interface Automata and unlike IO Automata, the TinyVT automaton is not required to be input-enabled, that is, $A_{in}(s) = A_{in}$ need not necessarily hold for any state $s \in S$. For a state $s \in S$, $A_{in} \setminus A_{in}(s)$ denotes the set of illegal inputs, that is, input actions that are not enabled when the current state is s. Furthermore, $A(s) = \oslash$ is allowed, to allow for modeling a final state from which no transitions originate. Unlike Interface Automata, TinyVT automaton assigns lower priorities to input transitions than to non-input (i.e. output or internal) ones. As a result, a TA is input enabled only if there is at least one enabled input transition and there are no enabled output or internal transitions at the given state.

Notice that the above definition of TinyVT automata allows for modeling nondeterminism, since T is a relation, not a function. For instance, $(s_i, a, s_j) \in T$ and $(s_i, a, s_k) \in T$ are allowed to hold at the same time, meaning that, from state s_i, for action a, the next state is randomly chosen to be either s_j or s_k.

3.8.2 Compositionality of automata

Two TinyVT automata M_1 and M_2 are composable if for every $a \in A_1 \cap A_2$ either $a \in A_{1_{in}} \cap A_{2_{out}}$ or $a \in A_{1_{out}} \cap A_{2_{in}}$. This means, that every action that is shared between the two automata, denoted as $Shared(M_1, M_2) = A_1 \cap A_2$, is an input of one and and output of the other.

The composition of automata M_1 and M_2 is denoted as $M_1 \| M_2$, where $\|$ is the (parallel) composition operator. If M_1 and M_2 are composable, their composition $M_1 \| M_2$ is defined as

- $S_{M_1 \| M_2} = S_1 \times S_2$,
- $s_{0_{M_1 \| M_2}} = (s_{0_1}, s_{0_2})$,
- $A_{in_{M_1 \| M_2}} = A_{in_1} \cup A_{in_2} \setminus Shared(M_1, M_2)$,
- $A_{out_{M_1 \| M_2}} = A_{out_1} \cup A_{out_2} \setminus Shared(M_1, M_2)$,
- $A_{h_{M_1 \| M_2}} = A_{h_1} \cup A_{h_2} \cup Shared(M_1, M_2)$,
- $T_{M_1 \| M_2} = \{((s_{i_1}, s_{i_2}), a, (s_{j_1}, s_{i_2})) \mid (s_{i_1}, a, s_{j_1}) \in T_1 \ \wedge \ a \in A_1 \setminus A_2\}$
  $\cup \{((s_{i_1}, s_{i_2}), a, (s_{i_1}, s_{j_2})) \mid (s_{i_2}, a, s_{j_2}) \in T_2 \ \wedge \ a \in A_2 \setminus A_1\}$
  $\cup \{((s_{i_1}, s_{i_2}), a, (s_{j_1}, s_{j_2})) \mid (s_{i_1}, a, s_{j_1}) \in T_1 \ \wedge \ (s_{i_2}, a, s_{j_2}) \in T_2 \ \wedge \ a \in Shared(M_1, M_2)\}$.

The rule describing how the transition relation of the composition is computed consists of three parts. The fist and second rules describe that if an action is accepted by one of the components but not the other, a transition is generated for the composite that advances the state of the component that accepts the action but leaves the state of the other component unaltered. The third rule describes that if an action is accepted by both components (which can happen only if it is an input action to one and an output action to the other) the state of both components are advanced.

At a given state, it is possible that a shared action is output by one of the components but not accepted by the other. Such states are called *illegal states*. We do not explicitly exclude illegal states from the composition, for two reasons. First, it is convenient to define the transition relation structure of the composite without constraints, leaving it to the behavioral semantics to specify how the automaton behaves. Second, depending on the environment, illegal states may or may not be reachable. For example, a composition which contains illegal states will work reliably in an environment that never drives the composition into an illegal state.

3.8.3 AsmL model

TinyVT automaton

The specification of semantics of the TinyVT automaton is split into two parts. First, the static data structures are specified, then the dynamic (behavioral) semantics is defined. The specification of semantics is given in the AsmL language, which, beside serving as a formal specification of semantics, allows for simulating a TinyVT automaton or a network of TinyVT automata using the AsmL tools.

Static data model

States are modeled as an AsmL class with two members, an optional unique name of the state and a Boolean value indicating wether the state is the initial state.

```
class State
  const name as String
  const initial as Boolean
```

Figure 38: TA state.

Action is modeled as a String, which holds the name of the action.

Transition is modeled as an AsmL structure, with the source and destination states, as well as the action associated with the transition as members.

```
structure Transition
  const src as State
  const dst as State
  const action as String
```

Figure 39: TA transition.

A TinyVT automaton is modeled as an abstract class. The automaton's set of states, set of input, output and internal actions, as well as the transition relation are modeled as abstract properties. The concrete data sources for these properties can be provided by subclassing.

```
abstract class AbstractTA
  abstract property inputActions as Set of String
    get
  abstract property outputActions as Set of String
    get
  abstract property internalActions as Set of String
    get
  abstract property states as Set of State
    get
  abstract property transitions as Set of Transition
    get
```

Figure 40: The Abstract Data Model of TinyVT Automata.

Notice that the properties only have accessors (get), not mutators (set), hence they cannot be modified as the state of the machine evolves.

Behavioral model

The behavioral model is described by continuing the implementation of the above classes and abstract classes by specifying variables that are dynamic, i.e. the values of which are changing while the state of the automaton evolves, and by specifying methods that manipulate the data structures.

The Boolean variable `active` is included as a field in the State class, to indicate wether the given state is the current state of the automaton, the state set of which it belongs. Furthermore, a constructor is provided, which sets the `active` flag if the state is the initial state.

```
class State
  var active as Boolean
  State(name as String, initial as Boolean)
      active = initial
```

Figure 41: Behavioral aspect of TA state.

The dynamic behavior of the automaton is specified by the `Step` method. The `Step` method has an argument of type `String?`, which, when the argument value is not `null` (empty input), defines an input to the automaton and directs the automaton to execute a corresponding input transition. The value of the argument can be `null`, in which case the automaton can take an internal or an output transition. After the transition is executed, the corresponding action is available as the return value of the `Step` method.

The assertion `require Accepts(a)` is used to verify that the automaton accepts the action at the current state. The `EnabledTransitions` method computes the set of input transitions that are enabled at the current state for the input action given as parameter, or the set of enabled output or internal transitions, if the parameter is `null`. If multiple transitions are enabled, a nondeterministic choice is made to randomly select one. The selected transition is executed by clearing the active flag of the source state and setting the active flag of the destination state simultaneously in one AsmL step.

```
abstract class AbstractTA
  Step(a as String?) as String?
    require Accepts(a)
    let transition = any t | t in EnabledTransitions(a)
    step
      transition.src.active := false
      transition.dst.active := true
      return transition.action
```

Figure 42: Behavior of a TA step.

The helper methods of the AbstractTA class are implemented as follows. The `Accepts` method returns `true` if the set of enabled transitions for the action given as a parameter is nonempty, otherwise it returns `false`.

```
abstract class AbstractTA
  Accepts(a as String?) as Boolean
    return Size(EnabledTransitions(a)) > 0
```

Figure 43: Deciding acceptance of an input.

The `EnabledTransitions` methods constructs the the set of enabled input transitions at the current state for the input action given as a parameter, or the set of enabled output or internal transitions if the parameter is `null`. Notice that an input transition is enabled only if there are no output or internal transitions originating from the current state.

```
abstract class AbstractTA
  EnabledTransitions(a as String?) as Set of Transition
    if a = null
      return EnabledOutputTransitions()
             union EnabledInternalTransitions()
    else
      return EnabledInputTransitions(a)

  EnabledOutputTransitions() as Set of Transition
    return { t | t in transitions where t.src = CurrentState() and
             t.action in outputActions }

  EnabledInternalTransitions() as Set of Transition
    return { t | t in transitions where t.src = CurrentState() and
             t.action in internalActions }

  EnabledInputTransitions(a as String) as Set of Transition
    if Size(EnabledOutputTransitions()) = 0 and
             Size(EnabledInternalTransitions()) = 0

      return { t | t in transitions where t.src = CurrentState() and
             a in inputActions and t.action = a }
    else
      return {}
```

Figure 44: Querying enabled transitions.

The `CurrentState` method returns the current state by selecting the one from the state set with the `active` flag set. The assertion `require Size(activeStates) = 1` asserts that no more than one state is active at a time.

```
abstract class AbstractTA
  CurrentState() as State
    let activeStates as Set of State =
             { s | s in states where s.active = true }

    require Size(activeStates) = 1
    return any s | s in activeStates
```

Figure 45: Querying the current state.

Composition of automata

Composition of automata is modeled as an abstract class, which contains references to its components. To allow for modeling hierarchical composition of automata, a common practice in software design, the AsmL model uses the Composite pattern[30]. Properties and operations of the components that are used when specifying the data model or the behavioral semantics of the composite are factored out to an interface. Since a composite can also be a component of a composite which is higher in the composition hierarchy, both the automaton class and the composite class needs to implement this common interface.

Static data model

The data aspect of the `IAbstractTA` interface contains the properties necessary for specifying of behavioral semantics of composition. Notice that neither the set of states, nor the transition relation is exposed through this interface.

```
interface IAbstractTA
  property inputActions as Set of String
    get
  property outputActions as Set of String
    get
  property internalActions as Set of String
    get
```

Figure 46: The IAbstractTA interface specifies the Abstract Data Model of parts in a composition.

The `AbstractTA` class is modified incrementally to implement the `IAbstractTA` interface as follows.

```
abstract class AbstractTA implements IAbstractTA
```

Figure 47: TA implementing the IAbstractTA interface.

The data model of the composition is modeled as an abstract class. It implements the `IAbstractTA` interface. The parts of the composite are given as an abstract property of type `Set of IAbstractTA`, this way, components of a composite may be both automata and composites.

The sets of shared, internal, input and output actions are computed from the internal, input and output actions of the components, according to the definition of composition rules.

```
abstract class AbstractComposite implements IAbstractTA
  abstract property components as Set of IAbstractTA
    get
  property sharedActions as Set of String
    get
      return { a | c in components, a in c.inputActions }
             intersect { a | c in components, a in c.outputActions }
  property internalActions as Set of String
    get
      return { a | c in components, a in c.internalActions }
             union sharedActions
  property inputActions as Set of String
    get
      return { a | c in components, a in c.inputActions }
             - internalActions
  property outputActions as Set of String
    get
      return { a | c in components, a in c.outputActions }
             - internalActions
```

Figure 48: Abstract Data Model of TA composition.

Behavioral model

In the behavioral model, two methods are added incrementally to the `IAbstractTA` interface: `Accepts` and `Step`.

```
interface IAbstractTA
  Accepts(a as String) as Boolean
  Step(a as String?) as String
```

Figure 49: Behavioral aspect of the IAbstractTA interface.

The above two methods are implemented by the AbstractComposite class as follows. `Accepts` returns `true` if any of the components accept the input, internal or empty action in the current state. Building the state structure of the composite, which is the power set of the state sets of the parts, and computing the composite's transition relation would be a complex task in AsmL, because the type of the composite state can be an arbitrary n-tuple, where n is the number of leaf AbstractTA instances in the composition hierarchy. Instead, the return value of `Accepts` is computed on the fly by delegating the call to the `Accepts` methods of the components. This way, the state set and transition relation of the composite do not have to be explicitly computed.

If `Accept` is called with `null` as parameter, `true` is returned if any of the components have output or internal transitions enabled. If the parameter not `null`, but an input action, `Accepts` return `true` if no components have output or internal actions enabled and there is a component that accepts the input action, given as parameter, at the current state. In any other cases, `Accepts` returns `false`.

```
abstract class AbstractComposite implements IAbstractTA
  Accepts(a as String?) as Boolean
    if (a = null)
      return (exists c in components where c.Accepts(a))
    else
      if Accepts(null)
        return false
      else
        return (exists c in components where c.Accepts(a)
             and a in inputActions)
```

Figure 50: Deciding acceptance of an input in composition.

The `Step` method is broken into two parts, depending on wether the action, given as an argument indicates that an input transition or an output/internal transition is to be taken. If the argument is `null`, that is, an empty input, an output transition is taken, otherwise, an input transition is executed. The assertion `require Accepts(a)` guarantees that there exists a part that accepts the input action or empty input.

```
abstract class AbstractComposite implements IAbstractTA
  Step(a as String?) as String
    require Accepts(a)
    if a = null
      step
        return OutputOrInternalStep()
    else
      step
        return InputStep(a)
```

Figure 51: Behavior of the TA composition step.

As a reaction to an input action, the composite forwards the input action to the contained component the set of input actions of which contains the given action. The component takes the corresponding transition, and, as a result, the state of the composite also changes.

As a reaction to an empty input, an internal or an output transition is taken. First, a component is selected that accepts the empty input, i.e. has an output or internal transition enabled at the current state. Then, the `Step` method of the selected component is invoked, which causes the selected component to take either an internal or an output transition. If an internal transition of the component was taken, the `Step` method of the composite returns `null`. If the selected component took an output transition, where the output action is an output action of the composite, the output action is returned by the Step method of the composite. However, if the selected component took an output transition, but the resulting output action is a shared action, that is, it is an input to another component within the composite, the corresponding input action of the latter is taken within the same step of the Abstract State Machine. The output transition of the former and the input transition of the latter is executed simultaneously by the composite, taking the form of a single internal action.

```
abstract class AbstractComposite implements IAbstractTA
  InputStep ( a as String ) as String
    let cs as Set of IAbstractTA = { c | c in components
             where a in c.inputActions }
    require Size(cs) = 1
    choose c in cs
      return c.Step(a)

  OutputOrInternalStep () as String
      let cs as Set of IAbstractTA = { c | c in components
             where c.Accepts(null) }
      choose c in cs
        let outputAction = c.Step(null)
        if outputAction in sharedActions
          return InputStep(outputAction)
        else
          return outputAction
```

Figure 52: TA composition step helper methods.

For reference, the compete AsmL sources, along with a simple example on how to instantiate the abstract classes to simulate a composition of TAs is given in Appendix B.

According to the behavioral semantics of the composite described above, composition of two deterministic TAs is always deterministic.

When an input event is sent to the composite, it forwards the event to its component the input event set of which contains this event. This component can be uniquely identified, since an event can only be accepted by one component.

As a reaction to the event, the component takes a (potentially empty) series of internal transitions followed by an output transition. Since the component is assumed to be deterministic, the input uniquely determines the component's output and new state.

If the output of the component is an input to the other component, it is sent as an input to the other component, which in turn, will take a deterministic series of transitions, resulting in a new state and returning an output. This process continues until the output of one part is not an input to the other, in which case, it is output by the composite.

Since the composite does not interact with the environment while computing the output in response to an input, the component's state (which is a tuple consisting of the states of the components) is evolving according to the deterministic rules described above. Since the state uniquely specify the output, the output is deterministic, as well.

3.8.4 Mapping TinyVT threads to TinyVT automata

This section explains how TinyVT threads can be mapped to TinyVT automata. The mapping requires only static information, which is statically extractable from the source code of TinyVT threads and available as an abstract syntax tree and a static control flow graph. The target of the mapping is the static data model of the TinyVT automaton.

Function invocations and await statements are the only points where a thread may interact with its environment. A thread interacts with its environment by receiving and passing control (optionally along with some data) from and to its environment, respectively. There are four types of such interactions:

- When a thread is *block*ed, it can receive control and resume executing after receiving an event (a function call) from the environment. This is expressed in TinyVT as an await statement with an inlined event handler.
- When a thread *yield*s, control is passed back to the source of the awaited event that triggered the actual execution context when the control reaches the next await statement.
- Threads may *call* out to external functions. While the external function is executing, the thread temporarily relinquishes control to the implementation of the called function.
- When an external function is executing as a result of a call by the thread, the thread *wait*s until a return from the external function passes control back to the thread.

There are four kinds of thread state associated with these interactions:

- *Blocking state:* The thread is blocked and is waiting for an external event which will cause the thread to resume computation.
- *Yielding state:* The thread has reached the end of the current execution context which was triggered by the most recently accepted event and is ready to return control to the originator of that event.
- *Calling state:* Thread execution has reached an invocation of an external function and the thread is ready to pass control to the external function.
- *Waiting state:* The thread is waiting for the external function it has previously invoked to finish and return the control back to the thread.

Assumptions

For the sake of simplicity, let us assume that every await statement has exactly one inlined event handler. Furthermore, we will assume that control flow within a thread does not depend on the values of function parameters or return values or global, static or shared variables. This simplifying assumption allows us to model only the control flow aspect of the interactions between threads such that the the resulting TinyVT automata will always be deterministic. Later I will explain that the above simplifying assumptions can be relaxed, at the cost of increased complexity of mapping rules and increased model size.

States

The first await statement in the thread maps to a blocking state which is the initial state of the TA. All other await statements map to two states: a yielding state, an output transition from which returns control to the event that triggered the thread's current execution context, and a blocking state, an input transition from which will resume thread execution. The destination of transition from the yielding state is corresponding blocking state.

For every function invocation, two states are generated in the data model of the TA: a calling state, a transition from which will generate an output action and pass control to an external function, and a waiting state, an input transition from which represents the return from the function call. The destination of transition from the calling state is the corresponding waiting state.

Actions

Each awaited event maps to an input action and an output action. The input action, enabled at some blocking state, represents passing the control from the originator of the event to the thread, while the output action, enabled in some yielding state, represents the return to the caller.

For every external function that is invoked by the tread, an output action and an input action is generated. The output action, enabled at some calling state, corresponds to the function invocation, and the input action, enabled at some waiting state, to the return from the external function.

Transitions

For every blocking state, an input transition is generated, where the input action corresponds to the event that is specified in the await statement that maps to the blocking state. The destination state of the transition will be the next call or yield state in the control flow graph (whichever appears first).

For every yielding state, an output transition is generated. The output action corresponds to the event that that triggered the current execution context. The destination state of the transition is a blocking state that is the mapping of the same await statement as the yielding state.

We generate an output transition from every calling state. The output action corresponds to the external function being called, the destination state is the waiting state that is the mapping of the same function invocation as the calling state.

Finally, an input transition is generated for every waiting state. The input action corresponds to the return from the external function for which the thread is waiting, the destination state is the next call state or yield state in the control flow graph (whichever appears first).

The resulting TinyVT automaton is deterministic, since there is at most one out-transition from every state.

Relaxing the assumptions

The following paragraphs explain that threads can be mapped to TinyVT automata even if the initial simplifying assumptions are relaxed. Without these assumptions, the mapping will be more complex and the resulting automata will increase in size (number of states, as well as number of transitions), but determinacy is still guaranteed. It is important to note, however, that this increase in size and complexity of mapping is irrelevant when the specification of behavioral semantics is in focus: Once there is a mapping defined from thread to automaton, the behavioral semantics of the automaton will apply to the thread that is mapped to an automaton. Therefore, in the following paragraphs, I will argue that we *can* always give a mapping from thread to automaton, but the details on *how* we can efficiently give a mapping is irrelevant.

Await statements with multiple events

As a consequence of the initial assumptions, namely that every await statement has exactly one inlined event handler and that the threads are free from data dependencies, the output event on the transitions originating from a yielding state can be unambiguously computed. When the control reaches an await statement and the thread yields, it needs to yield (return control) to the originator of the triggering event of the actual execution context. However, if the first assumption is relaxed, and we allow for multiple inlined event handlers within an await statement, it is not possible to unambiguously tell what is the triggering event of the actual execution context, if the await statement that is the entry point of the current execution context has more than one triggering events.

To overcome this problem, the triggering event has to be encoded into the automaton's state. Instead of generating a single yielding state for an await statement, we generate as many yielding states as many different events the previous await statement has, and tag each of the yielding states with the event names. Similarly, the the event names of the preceding await statement are encoded in the calling and waiting states generated from function invocations. When generating the transitions, the destination state is chosen such that the event name tags of the source and destination states are identical. Blocking states have no event name tags: They are join points of branches with different event name tags.

Data dependencies

The mapping as described above fails to capture scenarios where the control flow of a thread depends not only on the type of events received from the environment, but also on some data values that are passed along with the events. A straightforward way of handling data dependencies of this kind is treating two events of the same type but of different data values as separate input actions. That is, for an await statement with one embedded event handler which has a 8-bit integer parameter, 256 different input actions will be generated. Similarly, for return values of external functions, a separate input action has to be generated for all possible values. The same technique should be applied to output actions, to model the different data values that are inputs to some other automata in a composition.

More sophisticated handling of data dependencies can be achieved by using predicate abstractions [5]. Instead of generating a separate action for every possible function argument and return value from external functions called by the thread, it is possible to identify sets of values for which the control flow of the thread is identical. These sets can be described with predicates over the function arguments and over return values from external functions called by the tread. It is sufficient to create one input action for every such set, reducing the number of input actions in the TA model.

Dependencies of control flow on static and shared variables can, for example, be handled by encoding the variable values in the states of the TinyVT automaton. A global variable that is read and written by multiple threads need to be factored out into a separate TA, and accessed with getters and setters.

3.9 Discussion

Formal specification of semantics is essential for programming languages and programming models. The lack of such a specification, or informal/incomplete specifications can lead to semantic ambiguities, often resulting in unexpected program behavior or system failure.

The existence of a formal specification of semantics will help the general acceptance of a language. Also, it is important for programmers, compiler writers and tool integrators, as well. Clearly, programmers need to know the exact meaning of the language phrases they use, while the developer of the compiler must ensure that the compiler adheres to the semantics of the language. Without a formal specification of semantics, these parties may have different assumptions on the language, leading ambiguous and incorrect programs.

In this chapter, I presented the formal semantics of TinyVT language and analyzed the compositional behavior of TinyVT threads. An important part of this work is the observation that, in contrast with library based threading approaches in C, it is possible to unambiguously define the semantics of TinyVT threads, since threads are mapped to single-threaded C code by the TinyVT compiler, and thus, the ambiguities observed by Boehm do not surface [9].

Language semantics

I formalized the the operational semantics of the TinyVT language using the Abstract State Machines (ASM) approach (formerly known as Evolving Algebras [34]). Gurevich and Huggins gave a formal semantics specification for the ANSI C language in [35], which has been used as a starting point in specifying the semantics of the TinyVT language. Since TinyVT is an extension of C, it was sufficient to describe the meaning of the new language constructs which TinyVT introduced, and altering the semantics of some C language constructs, the behavior of which is altered when used within TinyVT threads.

The Abstract State Machines approach has been an excellent vehicle to formalize the semantics of the new language constructs, because it allows for structuring the specification into different abstraction layers where each layer is a refinement of a higher-level one. Such a layering gives a better structure to the specification, and makes it possible to define the semantics of language features by omitting irrelevant details that hinder comprehension.

Specifically, the first abstraction layer of the specification describes the control flow semantics of a TinyVT thread. This layer captures that each thread (conceptually) has its own, independent thread of execution, the control flow of which is specified by the C control structures within the thread's source code. Blocking statements (`await` and `yield`) are handled as opaque statements at this abstraction level.

The fact that a thread's independent control flow is just an abstraction — which is provided by the language and the compiler — is only revealed in the fourth abstraction layer of the specification of semantics. The fourth layer describes C function definitions and function invocations. Since TinyVT's `await` and `yield` statements are essentially calls to and returns from C functions, their semantics is also described here. Therefore, while the first layer describes how control flow of a thread is perceived from the thread's point of view, the fourth layer specifies that, from the outside, the environment perceives the thread as a set of function definitions that implement event handlers.

Compositional semantics

I defined the compositional behavior of TinyVT threads by following the semantic anchoring approach developed by Chen et al. [14]. First, I defined a finite automata based model (referred to as a *semantic unit* in Chen's terminology), called TinyVT automata (TA), which is sufficient to capture the interaction patterns between TinyVT threads, assuming that they are running on top of a pure event-driven runtime. TinyVT automata hide the details of the thread's computation by modeling it as a series of internal actions, however, it exposes the points where control flow leaves or returns to the thread by modeling them as output or input actions, respectively.

The specification of the TinyVT automaton is given in the AsmL language [37], and consists of two parts: structural (static) and behavioral (dynamic) semantics. The structural semantics specify the abstract data model of the automaton, which is, in this case a tuple including states, actions and transitions. The behavioral semantics define the operational rules of the automaton in terms of concepts defined in the structural semantics. I showed that a mapping is possible from TinyVT threads to the structural model of the TinyVT automaton, thereby establishing behavioral semantics for TinyVT threads. Specifically, I explained how the static structure of a thread, given as the static control flow graph, can be mapped to states, actions and transitions of the automaton. I described the compositional semantics — the structural and the behavioral aspects separately — of TinyVT automata in AsmL. Since the semantics of TinyVT threads can be anchored to that of TinyVT automata, the behavior of composite TinyVT automata reveals the semantics of systems composed of TinyVT threads.

I also showed that the resulting TinyVT automaton is always deterministic, and that determinism is preserved through composition. This implies that TinyVT threads, as well as compositions of TinyVT threads are always deterministic. In such a composition, the execution of conceptually concurrent TinyVT threads is interleaved, and the points of interleaving are always either yield points or function call sites. It is important to note, however, that these findings will not hold if the assumptions on the event-driven runtime are relaxed, for instance, by allowing asynchronous invocation contexts (e.g. interrupts) propagate into TinyVT threads.

Chapter IV

Conclusion and future work

In this work, I presented TinyVT, a compiler-assisted threading abstraction which enables programming event-driven software components as if they had their own, independent thread of execution. TinyVT bridges the gap between multithreading and the event-oriented programming model in the sense that it provides the intuitiveness and expressiveness of the former, while retaining the advantages of the latter — such as small memory footprint or the lack of need for locking. This section reiterates the contributions of my work, and highlights some future research directions in the realm of compiler-assisted concurrency abstractions.

4.1 Contributions

Thread abstraction for event-driven systems

The novelty of this work is that TinyVT provides language support to describe event-based computation with threads, in a well structured, linear fashion, without compromising the expressiveness of the implementation language. TinyVT's thread abstraction is transparent: the underlying event-driven execution model remains exposed to the programmer, therefore, both threads and event-driven code may coexist within an application. Since TinyVT is an extension to the C programming language, mixing TinyVT threads and C code is allowed. The TinyVT compiler will only process the code within TinyVT threads, leaving any event-driven C code unmodified.

Automated management of control flow

Event-driven programs consist of a set of event handlers, but their logical sequentiality cannot be described without explicit language support. Therefore, programmers need to implement event-driven applications as explicit state machines, manually managing the control flow. The abstraction of a thread that TinyVT introduces is a simple language extension that provides a means to express linear control flow in event driven programs, using C control structures (`if`, `while`, etc.) and blocking operations.

While TinyVT's thread abstraction helps automate tedious and error prone tasks in event-oriented programming, it does not hide the event-driven nature of the applications. In fact, the syntax of TinyVT requires that the programmer explicitly specify yield points in a thread, and guarantees that thread execution never blocks between yield points. This feature ensures that the programmer is aware of the control flow between conceptually concurrent threads. Calls to functions external to the thread explicitly state which thread the control is passed to; similarly, TinyVT's `await` statement is used to explicitly specify 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).

Compiler-managed allocation of local variables

TinyVT's most important asset is that the compiler automates the tasks that programmers traditionally do by hand: manual control flow management and manual stack management. As the complexity of applications keeps growing — and this is what is happening in the WSN domain —, such tasks are becoming increasingly hard to manage in the presence of severe resource-constraints.

TinyVT allows for declaring variables that are shared between event handlers as local variables using C's scoping rules. The compiler identifies these declarations and allocates the variables to static memory. By analyzing the structure of nested scopes in TinyVT threads, the compiler may assign multiple variables to the same memory region if the scopes of those variables are never active concurrently. This intelligent variable allocation is, in fact a compile-time, static emulation of the C stack, in compliance with the semantics of C's automatic storage duration.

For a reasonably complex event-driven program, memory efficient manual stack emulation is a prohibitively complicated task. However, the TinyVT compiler can easily cope with this complexity, and thus, produce better quality and more reliable code than an average programmer.

Formal specification of language semantics

An entire chapter of this dissertation is dedicated to the semantics of TinyVT threads. Although its importance is often understated, it is imperative that the semantics of a programming language be formally specified. The lack of a specification, or an informal or incomplete one can lead to semantic ambiguities that often manifest themselves in system failures or undesired behavior.

I provide the operational semantics of TinyVT's language constructs using the Abstract State Machines (ASM) formalism (formerly known as Evolving Algebras [34]), by building on an existing formal semantics specification of C [35].

In [9], Boehm showed that it is not possible to unambiguously specify the semantics of multithreaded programs implemented in the C language using threading libraries. An important finding of this work is that, in contrast to library based threading approaches in C, the compositional semantics of TinyVT *can* be specified.

To help specifying the compositional semantics of TinyVT threads, I present a semantic unit, called TinyVT automata, the *structural*, *behavioral* and *compositional semantics* of which is specified using the ASM formalism in the AsmL language [37]. I show that the static structure of TinyVT threads (precisely, the local control flow graph) can be mapped to the structural specification of a TinyVT automaton. This way, the behavioral and compositional semantics of TinyVT automata will directly apply to the threads, as well.

4.2 Future work

New language features

A possible future research direction is introducing new language features to improve the expressiveness of the TinyVT language. Currently, it is not possible to define (and redefine) default behavior in response to input events: for all event kinds that are accepted by a blocking thread, an event handler must be explicitly specified within the corresponding `await` statement. A syntactic shortcut, similar to the try-catch-finally construct in modern procedural languages, could alleviate this requirement and would allow for cleaner, less verbose TinyVT sources.

Currently, TinyVT applies C scoping rules to local variables. Therefore, parameter values and local variables of an event handler cannot be accessed from the code following the `await` statement in which the event handler is inlined. In such cases, information must be shared using static, global or compiler-managed local variables. However, extending the TinyVT language with a feature that allows information sharing on the C stack could improve the overall memory usage of the applications.

A generalized blocking statement could allow for controlled reentrance in TinyVT threads, a feature that is currently not available. Call sites in threads — which are currently defined using ANSI C's function call expressions — could be separated to a function invocation and waiting for the return value, allowing for incoming events in between the two. This way, a thread could accept and service incoming events while it has a function call pending, which is a recurring pattern in event-driven services.

Support for asynchronous events

The C code generated by the TinyVT compiler has minimal requirements on the underlying event-driven runtime, one of them being that all event handler invocations must originate – directly or indirectly – from the dispatcher, which is assumed to be single-threaded. In practice, especially when working close to the hardware-software boundary, it is often desirable to allow asynchronous interrupt contexts to call into the event-driven code. One possible point of improvement is relaxing this assumption, such that we only require the runtime to guarantee that no events are sent to the thread unless the thread is blocked. This change would allow interrupt contexts call into TinyVT threads, and it would also permit using schedulers that may have more than one event handlers executing at at time (e.g. with time slicing).

Such a small change in assumptions, however, would imply an avalanche of nontrivial changes to the language, compiler and, most importantly, to the compositional semantics of TinyVT threads. A desirable new language feature would be a specifier for an event handler definition, which could be used to express that the event has no effect on the control flow of the thread. Therefore, although an ongoing computation could be interrupted, race conditions on defining control flow could be prevented. Also, the compiler must be changed such that it generates reentrant, thread safe code, which would include some sort of a locking mechanism to provide mutually exclusive access to internal data structures (thread state and flags).

The compositional semantics of TinyVT threads would drastically change if asynchrony is allowed in TinyVT. The granularity of interleavings of threads would decrease to the level of binary machine instructions, leading to obstacles to creating a sound specification of semantics, as observed by Boehm in [9].

With proper locking and synchronization, however, TinyVT threads with support for asynchrony could have a whole new range of use cases, such as programming kernel services or device drivers in traditional operating systems, or implementing an entire OS using TinyVT threads, along the lines of Contiki [20] or TinyOS [44].

Whole-program analysis

Whole-program analysis techniques have the potential for additional gains in the performance and resource usage of TinyVT programs. The prototype TinyVT compiler processes thread definitions separately, generating disjoint blocks of C code (function definitions and variable declarations) from individual threads. However, analysis of scope structures and inter-thread communication patterns could possibly reveal that a pair of variables, each declared in a different thread, are never active at the same time, hence allowing for more aggressive compiler-managed variable allocation algorithms.

This is a very promising research direction, since this information is already captured in TinyVT programs. In contrast, such a memory allocation scheme would be very complicated to implement in traditional event driven systems with no explicit support for the definition of control flow involving multiple event handler invocations.

Appendix A
Sample TinyVT Source and the Generated C Code

In this appendix, I present a sample TinyVT program, and the C code that corresponds to the output of the TinyVT compiler. To improve comprehension of the generated code, a simplified version is presented (mangling of declarators with the thread's name is removed, preprocessor-generated lines are discarded, etc.).

A.1 Sample TinyVt source code

```
thread sample_thread {
  printf("__block0\n");

  await( char myEvent() {
   return 'a';
  });

  printf("__block1\n");

  printf("__block2\n");

  while(1) {
   printf("__block3\n");
   await( char myEvent() {
    return 'b';
   });
  }

// await(); implicit
}
```

A.2 Compiler output (simplified)

```
typedef uint8_t __state_t;
__state_t __state;

typedef __state_t __yield_point_t;

inline __yield_point_t __get_yield_point() {
  return __state >> 2;
}

inline void __set_yield_point(__yield_point_t yp) {
  __state = (yp << 2) | (__state & 3);
}

enum {
  EXECUTING_MASK = 1,
  YIELD_POINT_MASK = 2
};

inline bool __is_set_flag(__state_t flag_mask) {
  return __state & flag_mask;
}

inline void __set_flag(__state_t flag_mask) {
  __state |= flag_mask;
}

inline void __clear_flag(__state_t flag_mask) {
  __state &= ~flag_mask;
}

// ----------------------------
typedef uint16_t __next_block_t;
enum {YIELD_BLOCK = 0};

// ----------------------------
// block implementations (returning next block id)

__next_block_t __block0() {
  printf("__block0\n");
  __set_yield_point(1);
  return YIELD_BLOCK;
}

__next_block_t __block1() {
  printf("__block1\n");
  return 2;
}

__next_block_t __block2() {
  printf("__block2\n");
  if(1)
    return 3;
  else
    return 5;
}

__next_block_t __block3() {
  printf("__block3\n");
  return 4;
}

__next_block_t __block4() {
  __set_yield_point(2);
  return YIELD_BLOCK;
}

__next_block_t __block5() {
  __set_yield_point(3);
  return YIELD_BLOCK;
}

// ----------------------------
// dispatch next block (specified by id of block)
__next_block_t __dispatch_next_block(__next_block_t next___block) {
  switch (next___block) {
    case 0: return __block0();
    case 1: return __block1();
    case 2: return __block2();
    case 3: return __block3();
    case 4: return __block4(); // yield hitting await 2
    case 5: return __block5(); // yield hitting await 3
    default: return YIELD_BLOCK;
  }
}


// ----------------------------
// inlined event handlers
```

```
char __myEvent_await1() {
  char __rval;
  {
    printf("myEvent inlined in await1\n");
    __rval = 'a';
  }
  __next_block_t next___block = 1;
  while((next___block = __dispatch_next_block(next___block)) != YIELD_BLOCK);
  __clear_flag(EXECUTING_MASK);
  return __rval;
}

char __myEvent_await2() {
  char __rval;

  printf("myEvent inlined in await2\n");
  __rval = 'b';

  __next_block_t next___block = 2;
  while((next___block = __dispatch_next_block(next___block)) != YIELD_BLOCK);
  __clear_flag(EXECUTING_MASK);
  return __rval;
}

char __myEvent_halt() {
  printf("Unexpected myEvent\n");
  while(1);
  return myEvent_halt();
}

// ----------------------------
// event handler stubs
char myEvent() {
  if(__is_set_flag(EXECUTING_MASK))
    return myEvent_halt();
  __set_flag(EXECUTING_MASK);
  switch(__get_yield_point()) {
    case 1: return __myEvent_await1();
    case 2: return __myEvent_await2();
    default: return __myEvent_halt();
  }
}

void initEvent() {
  __next_block_t __next_block = 1;
  __set_flag(EXECUTING_MASK);
  while((__next_block = __dispatch_next_block(__next_block)) != YIELD_BLOCK);
  __clear_flag(EXECUTING_MASK);
}
```

Appendix B

Formal Semantics of TinyVT Automata in AsmL

The structural, behavioral and compositional semantics of TinyVT automata are described using the ASM approach in the AsmL language. Below, I provide the corresponding AsmL sources — a collection of AsmL structures and abstract classes. Also, a sample code that demonstrates how these abstract classes can be instantiated to allow for simulating a composition of TAs is presented.

B.1 Static, behavioral and compositional semantics

```
// Abstract data model
class State
  const name as String
  const initial as Boolean

structure Transition
  const src as State
  const dst as State
  const action as String

abstract class AbstractTA
  abstract property inputActions as Set of String
    get
  abstract property outputActions as Set of String
    get
  abstract property internalActions as Set of String
    get
  abstract property states as Set of State
    get
  abstract property transitions as Set of Transition
    get

interface IAbstractTA
  property inputActions as Set of String
    get
  property outputActions as Set of String
    get
  property internalActions as Set of String
    get

abstract class AbstractTA implements IAbstractTA

abstract class AbstractComposite implements IAbstractTA
  abstract property components as Set of IAbstractTA
    get
  property inputActions as Set of String
    get
      return { a | c in components, a in c.inputActions }  - internalActions
  property outputActions as Set of String
    get
      return { a | c in components, a in c.outputActions } - internalActions
  property internalActions as Set of String
    get
      return { a | c in components, a in c.internalActions } union sharedActions
  property sharedActions as Set of String
```

```
    get
      return { a | c in components, a in c.inputActions }
             intersect { a | c in components, a in c.outputActions }

// Abstract behavioral model
class State
  var active as Boolean
  State(name as String, initial as Boolean)
      active = initial

abstract class AbstractTA
  Step(a as String?) as String
    require Accepts(a)
    let transition = any t | t in EnabledTransitions(a)
    step
      transition.src.active := false
      transition.dst.active := true
      return transition.action

  CurrentState() as State
    let activeStates as Set of State = {s | s in states where s.active = true }
    require Size(activeStates) = 1
    return any s | s in activeStates

  Accepts(a as String?) as Boolean
    return Size(EnabledTransitions(a)) > 0

  EnabledTransitions(a as String?) as Set of Transition
    if a = null
      return EnabledOutputTransitions() union EnabledInternalTransitions()
    else
      return EnabledInputTransitions(a)

  EnabledOutputTransitions() as Set of Transition
    return { t | t in transitions where t.src = CurrentState() and
             t.action in outputActions}

  EnabledInternalTransitions() as Set of Transition
    return { t | t in transitions where t.src = CurrentState() and
             t.action in internalActions}

  EnabledInputTransitions(a as String) as Set of Transition
    if Size(EnabledOutputTransitions()) = 0 and Size(EnabledInternalTransitions()) = 0
      return { t | t in transitions where t.src = CurrentState() and
             a in inputActions and t.action = a}
    else
      return {}

  WriteCurrentState()
    Write(CurrentState().name + " ")

interface IAbstractTA
  Accepts(a as String) as Boolean
  Step(a as String?) as String
  WriteCurrentState()

abstract class AbstractTA implements IAbstractTA

abstract class AbstractComposite implements IAbstractTA

  Step(a as String?) as String
    require Accepts(a)
    if a = null
      step
        return OutputOrInternalStep()
    else
      step
        return InputStep(a)
  // if input event is not null, the event is sent to the TA that
  // accepts it, and null will be returned
  InputStep ( a as String ) as String
    let cs as Set of IAbstractTA = { c | c in components where a in c.inputActions }
    // assert that there's only one such component
    require Size(cs) = 1
    choose c in cs
      return c.Step(a)

  // if input event is null, a TA which can move (i.e. can take an output
  // or an internal step) will step, if the output is consumed by another
  // TA, both will step, if not, the output will be returned
  OutputOrInternalStep () as String
      let cs as Set of IAbstractTA = { c | c in components where c.Accepts(null) }
      choose c in cs
        let outputAction = c.Step(null)
        if outputAction in sharedActions
          return InputStep(outputAction)
        else
          return outputAction

  Accepts(a as String?) as Boolean
    if (a = null)
      return (exists c in components where c.Accepts(a))
    else
      if Accepts(null)
        return false
      else
        return (exists c in components where c.Accepts(a) and a in inputActions)

  WriteCurrentState()
    step foreach ta in components
      ta.WriteCurrentState()
      Write(" ")
    WriteLine("")
```

B.2 Example of a concrete system

The following code illustrates an instantiation of the abstract classes that define structural and behavioral semantics of TinyVT automata and composition of TinyVT automata.

```
// concrete classes
class TA extends AbstractTA
  var states_ as Set of State
  var inputActions_ as Set of String
  var outputActions_ as Set of String
  var internalActions_ as Set of String
  var transitions_ as Set of Transition

  override property states as Set of State
    get
      return states_
  override property inputActions as Set of String
    get
      return inputActions_
  override property outputActions as Set of String
    get
      return outputActions_
  override property internalActions as Set of String
    get
      return internalActions_
  override property transitions as Set of Transition
```

```
    get
      return transitions_

class Composite extends AbstractComposite
  var components_ as Set of IAbstractTA
  override property components as Set of IAbstractTA
    get
      return components_

class MySystem
  var conf as Composite
  // send an input to the composite and drive by feeding in nulls
  // until an output is returned
  React(e as String) as String
    var a as String
    step
      a := conf.Step(e)
      conf.WriteCurrentState()
    step until a in conf.outputActions
      a := conf.Step(null)
      conf.WriteCurrentState()
    step
      conf.WriteCurrentState()
    step
      return a

Main()
  step
    // actions
    onReq = "onReq"
    onResp = "onResp"
    startReq = "startReq"
    startResp = "startResp"
    // states of TA0
    s0_0 = new State("s0_0", true)
    s0_1 = new State("s0_1", false)
    s0_2 = new State("s0_2", false)
    s0_3 = new State("s0_3", false)
    // transitions of TA0
    t0_01 = Transition(s0_0,s0_1,startReq)
    t0_12 = Transition(s0_1,s0_2,onReq)
    t0_23 = Transition(s0_2,s0_3,onResp)
    t0_30 = Transition(s0_3,s0_0,startResp)
    // instantiate TA0
    ta0 = new TA({s0_0, s0_1, s0_2, s0_3},{"startReq","onResp"},{"onReq","startResp"},
             {},{t0_01,t0_12,t0_23,t0_30})
    // states of TA1
    s1_0 = new State("s1_0", true)
    s1_1 = new State("s1_1", false)
    // transitions of TA1
    t1_01 = Transition(s1_0,s1_1,onReq)
    t1_10 = Transition(s1_1,s1_0,onResp)
    // instantiate TA1
    ta1 = new TA({s1_0, s1_1},{"onReq"},{"onResp"},{},{t1_01,t1_10})
    // instantiate the composition of TA0 and TA1
    conf = new Composite({ta0,ta1})
    // instantiate the system
    sys = new MySystem(conf)

    // start the system with startReq as an input
  step
    WriteLine("Input event: " + startReq)
    f = sys.React(startReq)
  step
    WriteLine("Output event: " + f)
```

References

[1] Abrach, H., Bhatti, S., Carlson, J., Dai, H., Rose, J., Sheth, A., Shucker, B., Deng, J., and Han, R. Mantis: system support for multimodal networks of in-situ sensors. *2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)* (2003), 50–59. 2.1.2, 2.7

[2] Adya, A., Howell, J., Theimer, M., Bolosky, W. J., , and Douceur, J. R. Cooperative task management without manual stack management. *Proceedings of the USENIX Annual Technical Conference* (2002), 289–302. 1.3.1, 2.1.2

[3] André, C. Representation and analysis of reactive behaviors: A synchronous approach. In *Proc. CESA 96* (jul 1996). 2.1.3

[4] Anlauff, M. Xasm - an extensible, component-based asm language. In *ASM '00: Proceedings of the International Workshop on Abstract State Machines, Theory and Applications* (London, UK, 2000), Springer-Verlag, pp. 69–90. 3.5.2

[5] Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. Automatic predicate abstraction of c programs. *SIGPLAN Not. 36*, 5 (2001), 203–213. 3.8.4

[6] Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Guernic, P. L., and de Simone, R. The synchronous languages 12 years later. *Proceedings of the IEEE 91*, 1 (2003), 64–83. 2.1.3

[7] Berry, G., and Gonthier, G. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming 19*, 2 (1992), 87–152. 2.1.3

[8] Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., and Han, R. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl. 10*, 4 (2005), 563–579. 1.2.1, 2.1.2, 2.7, 2.8.1

[9] Boehm, H. J. Threads cannot be implemented as a library. Tech. rep., Hewlett-Packard, nov 2004. 3.2, 3.9, 4.1, 4.2

[10] Börger, E. High level system design and analysis using abstract state machines. In *FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method* (London, UK, 1999), Springer-Verlag, pp. 1–43. 3.5

[11] Butler, Z., Corke, P., Peterson, R., and Rus, D. Networked cows: Virtual fences for controlling cows. *In Proc. of WAMES* (2004). 1.1

[12] Castillo, G. D. The asm workbench - a tool environment for computer-aided analysis and validation of abstract state machine models tool demonstration. In *TACAS 2001:*

Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (London, UK, 2001), Springer-Verlag, pp. 578–581. 3.5.2

[13] Cha, H., Choi, S., Jung, I., Kim, H., Shin, H., Yoo, J., and Yoon, C. The retos operating system: kernel, tools and applications. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks* (New York, NY, USA, 2007), ACM Press, pp. 559–560. 1.2.1, 2.1.1, 2.1.2

[14] Chen, K., Sztipanovits, J., and Neema, S. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software* (New York, NY, USA, 2005), ACM, pp. 35–43. 3.3, 3.9

[15] Cheong, E., Liebman, J., Liu, J., , and Zhao, F. Tinygals: A programming model for event-driven embedded systems. *Proceedings of the 18th Annual ACM Symposium on Applied Computing (SAC'03)* (mar 2003). 2.1.1

[16] Cheong, E., and Liu, J. galsc: a language for event-driven embedded systems. *Proceedigs of Design, Automation and Test in Europe 2* (2005), 1050–1055. 2.1.1

[17] Cook, J., Cohen, E., and Redmand, T. A formal denotational semantics for c. Tech. Rep. 409D, Trusted Information Systems, sep 1994. 3.4.1

[18] Cook, J., and Subramanian, S. A formal semantics for c in nqthm. Tech. Rep. 517D, Trusted Information Systems, oct 1994. 3.4.1

[19] de Alfaro, L., de Alfaro, L., and Henzinger, T. A. Interface automata. *SIGSOFT Softw. Eng. Notes 26*, 5 (2001), 109–120. 3.8.1

[20] Dunkels, A. *Programming Memory-Constrained Networked Embedded Systems.* PhD thesis, Swedish Institute of Computer Science, Feb. 2007. 1.2.3, 2.1.2, 4.2

[21] Dunkels, A., Finne, N., Eriksson, J., and Voigt, T. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems* (New York, NY, USA, 2006), ACM Press, pp. 15–28. 2.1.2

[22] Dunkels, A., Grnvall, B., and Voigt, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. *EmNetSI* (nov 2004). 2.1.2, 2.8.4

[23] Dunkels, A., Schmidt, O., and Voigt, T. Using protothreads for sensor node programming. *The Workshop on Real-World Wireless Sensor Networks* (jun 2005). 2.1.2

[24] Dutta, P., Grimmer, M., Arora, A., Bibyk, S., and Culler, D. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. *In Proc. of IPSN/SPOTS* (Apr. 2005). 1.1

[25] Edwards, S. A. *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. PhD thesis, University of California, Berkeley, 1997. 2.1.3

[26] Engler, D. R., Kaashoek, M. F., and O'Toole, J. Exokernel: An operating system architecture for application-level resource management. *Symposium on Operating Systems Principles* (1995), 251–266. 2.1.1

[27] Fok, C.-L., Roman, G.-C., and Lu, C. Mobile agent middleware for sensor networks: An application case study. In *Proc. of the 4th Int. Conf. on Information Processing in Sensor Networks (IPSN'05)* (April 2005), IEEE, pp. 382–387. 2.7

[28] for Standardization, I. O. *ISO/IEC 9899-1999, Programming Languages – C*. 1999. 2.5

[29] Gajski, D. D., and Ramachandran, L. Introduction to high-level synthesis. *IEEE Design and Test of Computers 11*, 4 (oct/dec 1994), 44–54. 2.1.3

[30] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, nov 1994. 3.8.3

[31] Gay, D., Levis, P., and Culler, D. Software design patterns for tinyos. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems* (New York, NY, USA, 2005), ACM Press, pp. 40–49. 1.2.3

[32] Gay, D., Levis, P., v. Behren, R., Welsh, M., Brewer, E., and Culler, D. The nesc language: A holistic approach to networked embedded systems. *SIGPLAN* (2003). 1.2.1, 1.2.3, 1.5, 2.1.1, 2.1.4, 2.6.1

[33] Gu, L., and Stankovic, J. A. t-kernel: providing reliable os support to wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems* (New York, NY, USA, 2006), ACM Press, pp. 1–14. 1.2.1, 1.2.2

[34] Gurevich, Y. Evolving algebras 1993: Lipari guide. 9–36. 1.5, 1.6, 3.5, 3.9, 4.1

[35] Gurevich, Y., and Huggins, J. K. 1.6, 3.4.1, 3.6, 1, 3.6.1, 3.9, 4.1

[36] Gurevich, Y., Rossman, B., and Schulte, W. Semantic essence of asml. *Theor. Comput. Sci. 343*, 3 (2005), 370–412. 3.5.2

[37] Gurevich, Y., Schulte, W., and Veanes, M. Toward industrial strength abstract state machines. Tech. Rep. MSR-TR-2001-98, Microsoft Research, oct 2001. 1.5, 1.6, 3.5.2, 3.9, 4.1

[38] Guttag, J. V., and Horning, J. J. The algebraic specification of abstract data types. *Acta Inf. 10* (1978), 27–52. 3.1.5

[39] G.Wener-Allen, Johnson, J., Ruiz, M., Lees, J., and Welsh, M. Monitoring volcanic eruptions with a wireless sensor networks. *In Proc. of EWSN* (2005). 1.1

[40] Han, C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. A dynamic operating system for sensor nodes. *In Proceedings of the 3rd international Conference on Mobile Systems, Applications, and Services* (jun 2005), 163–176. 1.3.2, 1.5, 2.1.1, 2.7

[41] Handziski, V., J.Polastre, J.H.Hauer, C.Sharp, A.Wolisz, and D.Culler. Flexible hardware abstraction for wireless sensor networks. In *Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN 2005)* (2005). 1.2.3

[42] Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3 (June 1987), 231–274. 2.1.3

[43] Hill, J., and Culler, D. Mica: a wireless platform for deeply embedded networks. *IEEE Micro 22*, 6 (2002), 12–24. 1.1

[44] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., , and 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)* (nov 2000). 1.2.1, 1.3.2, 1.5, 2.1.1, 2.6.1, 2.8.4, 4.2

[45] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (1969), 576–580. 3.1.5

[46] Kasten, O., and Rmer, K. Beyond event handlers: Programming wireless sensors with attributed state machines. *The Fourth International Conference on Information Processing in Sensor Networks (IPSN)* (apr 2005). 1.2.1, 2.1.3

[47] Koshy, J., and Pandey, R. Vmstar: synthesizing scalable runtime environments for sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems* (New York, NY, USA, 2005), ACM Press, pp. 243–254. 1.2.3, 2.1.2, 2.7

[48] Kothari, N., Gummadi, R., Millstein, T., and Govindan, R. Reliable and efficient programming abstractions for wireless sensor networks. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)* (jun 2007). 2.1.4

[49] Laurer, H. C., and Needham, R. M. On the duality of operating system structures. *SIGOPS Operating Systems Review 13*, 2 (1979), 3–19. 1.3.1, 2.1.2

[50] Lédeczi, A., Nádas, A., Völgyesi, P., Balogh, G., Kusý, B., Sallai, J., Pap, G., Dóra, S., Molnár, K., Maróti, M., and Simon, G. Countersniper system for urban warfare. *ACM Transactions on Sensor Networks 1*, 1 (Nov. 2005), 153–177. 1.1, 1.2.1

[51] Lédeczi, A., Völgyesi, P., Maróti, M., Simon, G., Balogh, G., Nádas, A., Kusý, B., Dóra, S., and Pap, G. Multiple simultaneous acoustic source localization in urban terrain. *In Proc. of IPSN* (Apr. 2005). 1.1

[52] Lee, E. What's ahead for embedded software? *IEEE Computer* (sep 2000), 16–26. 1.2.3, 1.3.1, 2.1.2

[53] Lee, E. The problem with threads. *IEEE Computer* (feb 2006), 33–42. 1.3.1, 2.1.2

[54] Levis, P., and Culler, D. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2002), ACM Press, pp. 85–95. 1.2.3, 2.1.2

[55] Levis, P., Gay, D., and Culler, D. Active sensor networks. In *Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)* (may 2005). 1.2.3, 2.1.2, 2.7

[56] Liu, H., Roeder, T., Walsh, K., Barr, R., and Sirer, E. G. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services* (New York, NY, USA, 2005), ACM, pp. 149–162. 2.7

[57] Lynch, N. A., and Tuttle, M. R. Hierarchical correctness proofs for distributed algorithms. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing* (New York, NY, USA, 1987), ACM, pp. 137–151. 3.8.1

[58] Madden, S., Franklin, M., Hellerstein, J., and Hong, W. Tag: a tiny aggregation service for ad-hoc sensor networks. In *5th Symposium on Operating Systems Design and Implementation (OSDI)* (dec 2002). 1.2.3

[59] Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst. 30*, 1 (2005), 122–173. 2.1.4

[60] McCartney, W. P., and Sridhar, N. Abstractions for safe concurrent programming in networked embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems* (New York, NY, USA, 2006), ACM, pp. 167–180. 2.7

[61] Mosses, P. D. *Action Semantics*, vol. 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992. 3.1.4

[62] Newton, R., Morrisett, G., and Welsh, M. The regiment macroprogramming system. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks* (New York, NY, USA, 2007), ACM Press, pp. 489–498. 2.1.4, 2.7

[63] Norrish, M. *C formalized in HOL*. PhD thesis, Cambridge University, 1998. 3.4.1

[64] Papaspyrou, N. *A Formal Semantics for the C Programming Language*. PhD thesis, 1998. 3.4.1

[65] Plotkin, G. D. A Structural Approach to Operational Semantics. Tech. Rep. DAIMI FN-19, University of Aarhus, 1981. 3.1.3

[66] Polastre, J., Szewczyk, R., and Culler, D. Telos: Enabling ultra-low power wireless research. *In Proc. of IPSN/SPOTS* (Apr. 2005). 1.1

[67] Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, D., and Jones, M. B. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON* (San Francisco, CA, USA, 1989), IEEE Comput. Soc. Press, pp. 176–178. 2.1.1

[68] Regehr, J., Reid, A., and Webb, K. Eliminating stack overflow by abstract interpretation. *Trans. on Embedded Computing Sys. 4*, 4 (2005), 751–778. 2.1.1

[69] Schmid, J. Executing asm specifications with asmgofer. Web page at: http://www.tydo.de/AsmGofer/, 1999. 3.5.2

[70] Sethi, R. A case study in specifying the semantics of a programming language. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1980), ACM, pp. 117–130. 3.4.1

[71] Shnayder, V., Hempstead, M., rong Chen, B., Allen, G. W., and Welsh, M. Simulating the power consumption of large-scale sensor network applications. In *SenSys* (2004), pp. 188–200. 1.2.2

[72] Simon, G., Maróti, M., Lédeczi, A., Balogh, G., Kusý, B., Nádas, A., Pap, G., Sallai, J., and Frampton, K. Sensor network-based countersniper system. In *In Proc. of ACM SenSys* (New York, NY, USA, 2004), ACM Press, pp. 1–12. 1.1, 1.2.1

[73] Slonneger, K., and Kurtz, B. L. *Formal syntax and semantics of programming languages.* Addison-Wesley Publishing Company, 1995. 3.1.2

[74] Stoy, J. E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, USA, 1981. 3.1.4

[75] Titzer, B. L. Virgil: objects on the head of a pin. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), ACM Press, pp. 191–208. 1.2.1

[76] v. Behren, R., Condit, J., and Brewer, E. Why events are a bad idea (for high-concurrency servers). *HotOS IX* (may 2003). 1.3.1, 2.1.2

[77] Volgyesi, P., Balogh, G., Nadas, A., Nash, C., and Ledeczi, A. Shooter localization and weapon classification with soldier-wearable networked sensors. *5th International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2007). 1.1, 1.2.1

[78] Welsh, M., and Mainland, G. Programming sensor networks using abstract regions. In *1st Symposium on Networked Systems Design and Implementation (NSDI 2004)* (mar 2004). 2.1.4

[79] Whitehouse, K., Sharp, C., Brewer, E., and Culler, D. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services* (New York, NY, USA, 2004), ACM Press, pp. 99–110. 1.2.3

[80] Yao, Y., and Gehrke, J. E. The cougar approach to in-network query processing in sensor networks. *ACM Sigmod Record 31*, 3 (sep 2002). 2.1.4