

# Inter-Context Control-Flow Graph for NesC, with Improved Split-Phase Handling

Arne Wichmann  
Institute for Software Systems

July 30, 2010

## Abstract

Control-flow graphs (CFG) are a common intermediate representation (IR) for a program and all the paths that might be traversed through its execution. But they are restricted to the intraprocedural case.

For sensor programs, it is important that the IR models the whole programs possible paths of execution, covering the interprocedural cases as well, because the programs are small enough to allow an analysis of the whole program and thorough testing is very important because changes after deployment are usually impossible. Recently, Lai, Cheung, and Chan introduced the inter-context flow graph (ICFG), which does model preemptions and resumptions of execution through interrupt handlers and covers the transitions of control-flow between tasks but does not cover the transitions of control-flow when normal functions are called and some of the transitions between tasks.

In this thesis, I present the inter-context control-flow graph (ICCFG), a new IR, which extends the ICFG with a model to cover normal function calls, proposes extended ways to construct the taskgraph and allows a selected component construction of the ICCFG, with the help of boundary information given by users. The ICCFG is implemented as a graph-extraction utility, which gives its output using the dot language and is integrated into the nesC compiler. Tests show that the ICCFG covers more of the existing control-flow of an applications than the ICFG does.



# Contents

<b>1. Starting Point</b>	<b>7</b>
1.1. TinyOS Operating System . . . . .	7
1.2. NesC Programming Language . . . . .	7
1.2.1. NesC Language Constructs (Syntax) . . . . .	8
1.2.2. TinyOS Idioms of NesC Use (Semantics) . . . . .	12
1.3. Inter-Context Flow Graph . . . . .	13
1.3.1. Intra-Procedural Control-Flow Graph . . . . .	14
1.3.2. Modelling Deferred Execution – Taskgraph . . . . .	15
1.3.3. Modelling Preemptive Execution . . . . .	16
<b>2. Inter-Context Control-Flow Graph</b>	<b>21</b>
2.1. Modelling Immediate Execution – Supergraph . . . . .	22
2.2. Modelling Deferred Execution – Taskgraph . . . . .	23
2.3. Improved Split-Phase Handling . . . . .	26
2.3.1. Direct Callbacks . . . . .	28
2.3.2. Indirect Callbacks . . . . .	28
2.4. The Blink Application . . . . .	31
<b>3. Formal Specification</b>	<b>33</b>
3.1. Definition of the Inter-Context Control-Flow Graph . . . . .	33
3.1.1. Basic Definitions . . . . .	33
3.1.2. Task Definitions . . . . .	34
3.1.3. Interprocedural Execution . . . . .	34
3.1.4. ICFG . . . . .	35
3.1.5. ICCFG . . . . .	35
3.2. Algorithms . . . . .	35
3.2.1. Task-Posting Tree Algorithm . . . . .	36
3.2.2. Taskgraph Construction Algorithms . . . . .	36
3.2.3. Taskgraph Fixpoint Iteration . . . . .	38
<b>4. Constructing the ICCFG</b>	<b>41</b>
4.1. Converting the Abstract Syntax Tree into the Control-Flow Graph . . . . .	42
4.2. Boundary Information . . . . .	44
4.3. Supergraph . . . . .	45
4.4. Generating the Dominator Tree for a Control Flow Graph . . . . .	46
4.5. Creating a Task-Posting Tree from a Dominator Tree . . . . .	46
4.6. Processing the Task-Posting Trees to Form a Taskgraph . . . . .	47

4.7. Traversing the ICCFG . . . . .	48
<b>5. Results</b>	<b>49</b>
5.1. The Testset . . . . .	49
5.2. ICCFGs for the Demo Applications . . . . .	53
5.3. Interpretation . . . . .	53
5.3.1. Implementation . . . . .	54
<b>6. Conclusion and Future Work</b>	<b>55</b>
<b>A. Appendix</b>	<b>57</b>
A.1. Blink Application Source . . . . .	57
A.2. RegionC . . . . .	58

## Introduction

Wireless sensor networks consist of several computational devices, called motes. The motes are deployed over the physical environment to be monitored. They may have one or more sensors attached to gather information. Some applications are temperature monitoring of large areas [1] or structural health monitoring of buildings by providing local informations about vibrations and material bending [2].

Many applications need very cheap and small motes. They may only cost cents and should be the size of a grain of sand. These shall then be distributed over areas and provide a cheap way to monitor its conditions. Most sensor networks need to transport data to one or more data sinks. The motes communicate via a radio-interface. Some use advanced routing protocols which allow them to build a dynamic self-organized network. These networks are usually capable of compensating the loss of single motes. Energy consumption of the motes poses a problem. The main source of energy often is a simple battery. The lifetime of batteries is much shorter than the lifetime of the mote itself. To save on hardware costs, many networks need regular visits of their motes by technicians for maintenance, especially to change the motes batteries. Development of self-powered solar motes exists [3] and for very small sensors energy sources like vibrations may be possible in the future [4].

Software running on the motes is bug prone, like any software. Testing and software analysis of applications before deployment is a very important topic for wireless sensor networks, because it is often difficult to change the software of the application in the field or to trace debug information in deployed motes. To assess the quality of software it is necessary to give certain metrics like code coverage or path coverage in a control-flow graph. For complex applications, simple intraprocedural analysis is no longer sufficient and new control-flow graphs have to be developed, which also cover control-flow between individual procedures.

Lai, Cheung and Chan [5] provide an inter-context flow graph (ICFG), which addresses the possible control flow between a set of tasks and addresses preemptive interrupt handling in TinyOS, an operating system for wireless sensors. They use two test-criteria to help analyze the coverage of their tests: All inter context edges and all inter context def-use pairs. All inter context edges is a control-flow criteria representing transitions of one context to another. All inter context def-use pairs is a data-flow criteria representing the flows of data between contexts. The ICFG does not cover direct calls between functions, it is not able to represent complex sequences of tasks, and it works only if all components of an application are analyzed.

On the other hand, the supergraph as presented by Reps, Horwitz and Sagiv [6] addresses the direct calling between functions.

The inter-context control-flow graph (ICCFG) presented in this work, joins the works on the inter-context flow graph and the supergraph. Its construction algorithms also honor the special cases, which arise if some of the applications components are left out of the constructed ICCFG and only stubs of these components are known. Those algorithms exploit that nesC, the language TinyOS is written in, allows abstract information about interfaces to be used to model hidden components

as blackboxes and model their behavior from a macroscopic view. The inter-context control-flow graph can be generated by a patched version of the nesC compiler.

This thesis is structured as follows:

Chapter 1 gives an overview about the starting point for this work. It shortly introduces the TinyOS operating system, explains the important aspects of the nesC language and explains the ICFG.

Chapter 2 introduces the new findings. It explains the flaws of the ICFG and introduces the ICCFG which addresses these limitations.

Chapter 3 gives a formal definition of the ICCFG to allow its used as a foundation of further analyses and gives a precise description of its construction. The algorithms used to construct the taskgraph in the ICFG and ICCFG are described and compared.

Chapter 4 shows what has been done. It explains how the ICCFG is constructed and describes the implementation.

Chapter 5 gives a resume of the work. It shows some basic tests and their results and describes ideas for future work.

# 1. Starting Point

This chapter gives an overview about the starting point for this work. It shortly introduces the TinyOS operating system, explains the important aspects of the nesC language needed to construct extended CFGs and informally explains the ICFG.

Size, price and energy limitations restrict nodes of wireless sensor networks to very simple controllers (no virtual memory, no memory protection, low clock speed) and small amounts of memory (few kilobytes). It is usually not possible to run modern, standard operating systems on them.

## 1.1. TinyOS Operating System

The most widely used operating system with a small memory footprint for nodes is TinyOS. It supports an event-driven programming model. Larger tasks can be executed deferred by a non-preemptive scheduler. The component features of nesC allow TinyOS to provide a large set of stock components, which an application simply can wire together to match its demands.

The TinyOS toolchain uses the nesC and GNU compilers. The nesC compiler (`nesc1`) reads all the necessary components for an application and compiles them to a single C file. The GNU compiler (`gcc`) with an appropriate backend (`avr,msp430`) compiles this file to the target binary. The use of a single C file as intermediate representation between the two compilers allows some optimizations that would not be possible with the `gcc` as second compiler otherwise (whole program optimization).

There exist two versions of TinyOS, version one was written in a mixture of plain C and nesC, where TinyOS version two is purely written in nesC. This thesis considers the features of TinyOS version two only. It only processes nesC components using the nesC compiler.

## 1.2. NesC Programming Language

NesC is a programming language for networked embedded systems. The development of nesC is mainly driven by David Gay (Intel Labs Berkeley) [7]. It enhances C with models for events, commands, components, and tasks. The language is a dialect of C, implemented as an extension to the C grammar (from Kernighan and Richie, C Programming Language, Second Edition [8]).

### 1.2.1. NesC Language Constructs (Syntax)

The nesC syntax elements which are of concern for this work are events, commands, function calls, interfaces, components, tasks and atomic blocks. These nesC syntax elements are explained in the following.

*storage-class-specifier*: also one of  
command event async task norace

*atomic-statement*:  
atomic statement

*call-kind*: one of  
call signal post

*comp-kind*:  
module  
configuration  
component  
generic module  
generic configuration

Figure 1.1.: Important NesC Grammar Rules

Figure 1.1 shows some of the additional grammar rules of nesC that are added to the standard C grammar. The figure shows all of the possibly control-flow altering rules of the nesC grammar. The `norace` keyword is not part of this work, because it only alters the warnings emitted by the nesC compiler. The `generic` modules and configurations are handled internally by the nesC compiler. The ICFG and ICCFG only process the already instantiated components. Both the `norace` and `generic` keywords are shown to give complete grammar rules from [9]. The `component` keyword is used to describe binary components and is therefore not part of the constructs covered in this work.

#### Event and Command as Special Function Calls

NesC commands and events form a special kind of function call, see `command` and `event` in *storage-class-specifier* from Figure 1.1. They are part of a components interface (see Listing 1.1). They differ from normal function calls, because they can have a fan-out. For example there may be more than one event handler registered for a single event. Normal functions are not part of nesC interfaces.

Commands can be called on the implementing component and events can be signaled from a component. See `call` and `signal` in *call-kind* from Figure 1.1.

The Example (Listing 1.1) shows the `Boot` and `Read` interfaces. The `Boot` interface signals the `booted` event, when the system is booted and ready to start the application code. The `Read` interface provides a `read` command, which is used to initiate a read process, which will return its result by signaling the `readDone` event.

```
interface Boot {
    event void booted();
}
interface Read {
    command void read();
    event void readDone(void *);
}
```

Listing 1.1: NesC Interface Definitions

## NesC Components and Interfaces

NesC uses components to hold the application's state and functionality. Components use other component's interfaces or provide implementations of interfaces on their own. Interfaces define sets of commands and events that must be implemented or bound to if one uses or provides an interface.

If users use an interface (of another component), they can call the contained commands on the other component and have to provide implementations bound to the events of the other component's interface.

If users provide an interface, they can signal the interfaces events and have to provide implementations for the commands to receive calls.

## NesC Modules

A nesC module defines a component and consists of two sections, a `module` and an `implementation` section. See `module` in *comp-kind* from Figure 1.1. The `module` section defines which interfaces are used or provided. The `implementation` section holds the actual code of the implementation.

In the example, from Listing 1.2, the component `Component1` uses the `Boot` interface and provides the `Read` interface, defined in Listing 1.1. It implements the `Boot` interface and therefore has to provide an implementation of the event handler `Boot.booted`. It provides the `Read` interface and has to provide the `read` command and can signal the `readDone` event.

```

module Component1 {
    uses interface Boot;
    provides interface Read;
}
implementation {
    void * data;

    event void Boot.booted() {
        // init component
    }

    command void read() {
        // acquire data
        signal readDone(data);
    }
}

```

Listing 1.2: Example Module Definition

## Wiring of Components

To configure the interaction of components, nesC uses configurations. See Figure 1.1, **configuration** in *comp-kind*. A configuration defines which components are actually used. In the **implementation** section the components are “instantiated.” It is possible to instantiate a component more than once by providing unique names for each of the instances.

For all interfaces of the components, the configuration provides a wiring that defines which component’s interface connects to which other component’s interface.

The example in Listing 1.3 instantiates the three components **MainC**, **Component1**, and **ReaderC**. **MainC** provides the **Boot** interface, which is wired to **Component1**’s **Boot** interface. The **Read** interface of **Component1** is wired to some **ReaderC** component.

A configuration may provide or use interfaces like a module itself in its configuration section. It is therefore possible to wrap inner components and provide an abstract interface for further use.

A top-level configuration, which does not provide or use interfaces itself, is used to give a wiring which defines a complete application.

## Preemptive Execution

NesC allows commands and events to be tagged as asynchronous (**async**). See **async** in *storage-class-specifier* from Figure 1.1. The asynchronous code can be run preemptively, stopping the normal path of execution and after the execution resuming it.

```

configuration BlinkAppC { }
implementation {
    components MainC, Component1, ReaderC;
    Component1 -> MainC.Boot;
    Component1.Read -> ReaderC;
}

```

Listing 1.3: BlinkApp Configuration Definition

Any command or event marked `async` is only allowed to call other commands or events that are marked `async`. It is allowed to call asynchronous functions from non-asynchronous code. The only way to leave an asynchronous context is to return from it by leaving all the asynchronous function bodies.

Asynchronous code is normally used to implement the handling of hardware interrupts. It is a common convention to keep asynchronous code very short, to be able to react to other interrupts in a short period of time. Long calculations for the received interrupt should instead be expressed as tasks, which are processed without inhibiting further interrupt handling.

The implementation of an asynchronous event handler is shown in Listing 1.4. It shows the possibility of `async` code to post a task `task1` to process its input data in a synchronized context. Listing 1.5 also implements an asynchronous event handler `f` and shows the infrastructure needed to implement mutual exclusion of `async` event handlers. The details of the example are explained in the section about atomic blocks.

### Deferred Execution with Tasks

NesC allows functions to be marked as tasks which may be posted. See `task` in *storage-class-specifier* and `post` in *call-kind* from Figure 1.1. Tasks are executed by a scheduler, which runs as the main routine of the application. If there are no tasks to execute available, the scheduler can put the application into a hardware standby state, if the hardware supports this or simply idle in a loop, until a new task arrives.

Any nesC code may post tasks for deferred execution.

The example in Listing 1.4 shows that the function `foo` is posting `task1`, so the scheduler will call the implementation of the task `task1`.

### Atomic Blocks

Statements and compound statements may be tagged atomic, see `atomic` in *atomic-statement* from Figure 1.1. Atomic blocks and statements are used to prevent pre-emption. Only non-atomic code may be interrupted by an `async` event or command. The current nesC implementation enforces atomic behavior by disabling all interrupts. Not all control structures are allowed in an atomic block. For example `goto`

```

async void foo() {
    post task1();
}
task void task1() {
    // do something
}

```

Listing 1.4: NesC Task Posting

may jump out of the block without reenabling interrupts and its use is therefore prohibited.

The example in Listing 1.5 shows the implementation of mutual exclusion with the use of atomic blocks for an asynchronous event handler. The asynchronous event handler `f` may be executed at any time of the program execution, even interrupting another execution of itself. To ensure that only one event handler is able to execute the function `do_sth`, the event handler uses atomic (uninterruptable) reads and writes for the boolean `busy` flag.

```

bool busy; // global
async event void f() {
    bool available;
    atomic {
        available = !busy;
        busy = TRUE;
    }
    if (available) do_sth();
    atomic busy = FALSE;
}

```

Listing 1.5: Atomic Block Example

### 1.2.2. TinyOS Idioms of NesC Use (Semantics)

This section will give an overview of typical nesC properties and idioms of language use in TinyOS Applications. It is important to know about them, because they allow this work to give a more precise ICCFG.

#### The Scheduler

TinyOS changed the semantics of its default scheduler from version one to version two. Version one uses a FIFO scheduler and allows tasks to be in the queue more than once. The queue has a fixed upper limit for its length. Posting can only fail if

the queue is full. This implies starvation of tasks in some circumstances and leads to the development of the version two model [10].

TinyOS version two provides a default scheduler that also implements a non-preemptive FIFO schedule. Posting of tasks uses a unique queue. This means that a task may only be in the queue once. A second post will not alter the queue and return information about the failed post to the user. Immediately before executing a task, it is taken out of the queue, allowing a repost of itself for further computation of its data and to provide fairness for other tasks [10].

TinyOS two also allows one to alter the semantics of task posting, execution and scheduling by providing user code to implement his own scheduler and subroutines processing the posting of tasks [10].

### Split-Phase Command and Event Pairs

TinyOS interfaces often have correspondences between some of their commands and events. The usual pattern consist of a command which starts some background work and an event, which is signaled when there is a result of the work. This pattern is called split-phase.

Some components with their typical interfaces are: A sensor interface implements a `read` command and provides a `readDone` event, given by the `Read` interface. Listing 1.1 describes the `Read` interface. A minimalistic implementation of the interface is given in Listing 1.2. It provides a `read` command that immediately signals `readDone`. A network protocol implements a `send` command and provides `sendDone` event, given by the `Send` interface. A periodic event generator implements a `startPeriodic` command and `stopPeriodic` command and provides a `fired` event, given by the `Timer` interface.

High-level components usually provide synchronized interfaces, which use background tasks invisible to the user which signal the events between the execution of user tasks.

The events provided by the split-phase operations may also be asynchronous events, which then behave like hardware interrupts. The `Alarm` interface provides such an `async fired` event for its encapsulated hardware timer. Some sensor interfaces provide a version of the `read` and `readDone` interface called `ReadNow` that also implements this asynchronous behavior.

## 1.3. Inter-Context Flow Graph

The inter-context flow graph, by Lai, Cheung and Chan [5], combines the intra-procedural control-flow graphs of single functions with interdependencies of tasks and preemptions and resumptions by interrupt handlers.

The ICFG is still important, although superseded by the ICCFG, because the ICCFG is an extension to the ICFG.

The following subsections will describe the intra-procedural control-flow graphs, used as the basis for the ICFG, the modelling of deferred execution with the taskgraph

and the preemption of execution by async event handlers.

This is only an informal discussion of the ICFG. Its formal definition is given in Section 3.

### 1.3.1. Intra-Procedural Control-Flow Graph

The control-flow graph (CFG) gives information about the control flow of a subroutine.

```
module Foo { }
implementation {
  int a,b; void A(); void B();
  void bar() {
    while (b) {
      if(a) A();
      else B();
    }
  }
}
```

Listing 1.6: Source Code to Figure 1.2

The nesC compiler provides an abstract syntax tree (AST) with nodes representing the grammar elements of the nesC syntax without any control-flow information. The simple AST does not give any direct information which statement will be executed before which other statement. For example (see Figure 1.2(a)) it is not possible to see that after the call to `B` the `while(b)` predicate will be executed next.

The CFG gives us this information. It is possible to see where the function can be left and how exactly the control flow looks. The example CFG (see Figure 1.2(b)) shows, that after the call to `B` the `while(b)` predicate will be executed. To create a CFG from an AST it is necessary to process all the nodes in the AST and use the information about the control-flow known for each of the statements in the AST. For example it is known for C-like languages, that after the body of a `while` loop was executed, its predicate will be checked again.

For constructing the taskgraph, a so called task-posting tree (TPT) is constructed. This construction is based on a dominator tree for every task. The dominator tree gives for all nodes in the CFG information about what nodes must have been executed before they are executed. The dominator tree (see Figure 1.2(c)) will help us answer questions like: “Is `B` always executed before the execution reaches the exit node?” In this example it can clearly be said “no”, because `B` is not a parent of the exit node, meaning that there are paths through the CFG from the entry to the exit node, not passing through `B`.

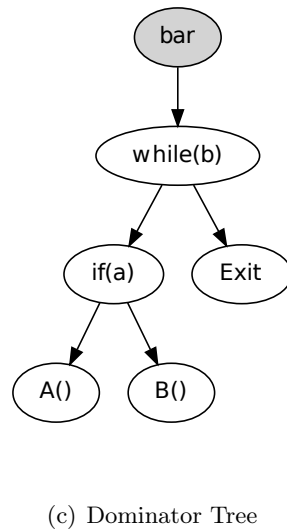
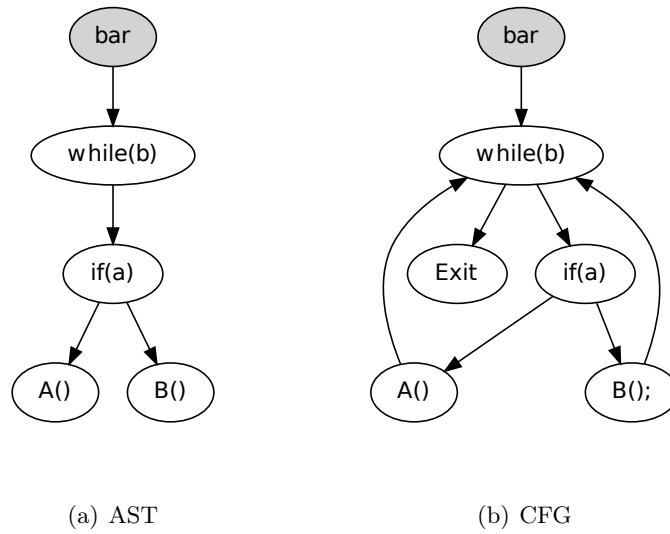


Figure 1.2.: AST, CFG and Dominator Tree of the subroutine `bar` listed in Listing 1.6

### 1.3.2. Modelling Deferred Execution – Taskgraph

Deferred Execution is a serial way of execution. The main information about the control flow is given through the possible order of tasks (or deferredly executed subroutines in general). The taskgraph, as proposed by Lai, Cheung and Chan extrapolates possible execution orders for tasks by processing the post-statements in the dominator trees of the subroutines of an application.

```

module task {
    uses interface taskI;
}
implementation {
    int a,b; void A(); void B();
    task void task3 () {};
    task void task2 () {
        while (b) {
            if(a) A();
            else B();
        }
    }
    task void task1 () {
        post task2 ();
        post task3 ();
    }
    event void taskI.eventhandler () {
        post task1 ();
    }
}

```

Listing 1.7: Taskgraph Example Source

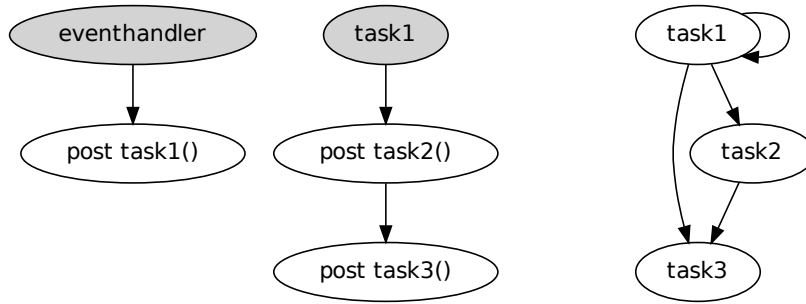
The task-posting trees are generated from the dominator trees of their respective subroutines. If the subroutine directly calls a function which does the actual task posting, additional processing is needed. These task-posting trees are used to create a taskgraph (see Figure 1.3(b)). Details about this process can be found in Section 3.2.2.

Figure 1.3(a) shows the task-posting trees of the subroutines of Listing 1.7. The `taskI.eventhandler` event will post the task `task1` for deferred execution. The `task1` itself will post the tasks `task2` and `task3`. This leads to the taskgraph shown in Figure 1.3(b). `task1` can be executed before any of its posts, and, because it is posted by an eventhandler, before itself. If `task2` is executed at all, it will always be executed prior to `task3`.

Figure 1.3(c) show the representation of the taskgraph as dashed edges together with the intraprocedural control-flow graphs of the subroutines. This is the first ICFG, representing all the possible control flow given in Listing 1.7. The only missing feature is preemptive execution.

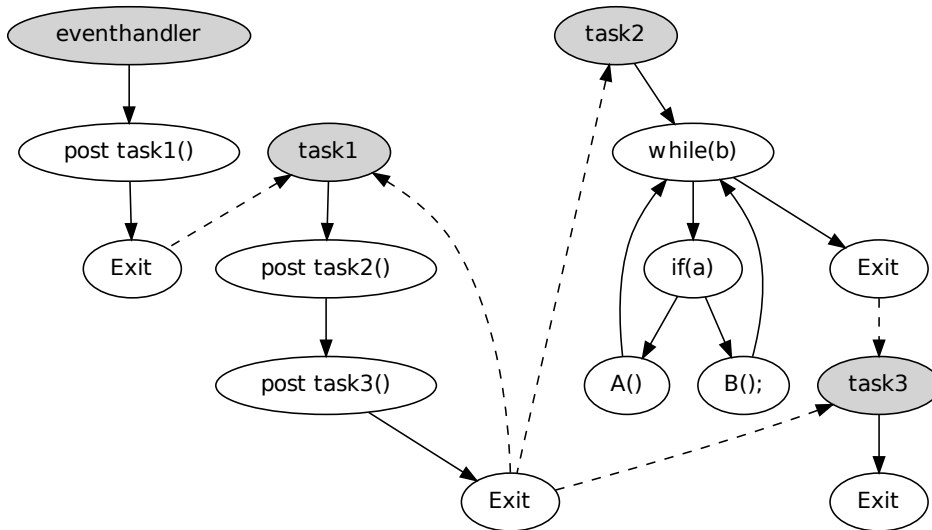
### 1.3.3. Modelling Preemptive Execution

Preemptive execution may interrupt the normal control flow at any point not marked atomic. The execution will return to the exact same point where the interruption



(a) Task-Posting Trees

(b) Taskgraph



(c) ICFG

Figure 1.3.: Taskgraph Example

occurred.

This behavior can easily be represented in the ICFG by adding edges from all non-atomic nodes to the entry node of the preemptively executable subroutine and from its exit node to all the non-atomic nodes.

To show a complete ICFG, the event `eventhandler` can be changed to be `async` (see Listing 1.8). This allows preemption and resumption to happen. Figure 1.4 shows the resulting ICFG. The dashed edges represent preemptions and resumptions.

```

module task {
    uses interface taskI;
}
implementation {
    int a,b; void A(); void B();
    task void task3 () {};
    task void task2 () {
        while (b) {
            if(a) atomic A();
            else atomic B();
        }
    }
    task void task1 () {
        post task2 ();
        post task3 ();
    }
    async event void taskI.eventhandler () {
        post task1 ();
    }
}

```

Listing 1.8: Preemptive Execution Example Source

It is possible to preempt every non-atomic, non-entry, and non-exit block. Atomic blocks are not preemptable, because the nesC implementation turns off the hardware interrupts, before the actual statement is executed. Entry and exit blocks are not preemptable, because they do not represent statements and only help to keep the graph readable. This explains why there are not many dashed edges in this ICFG.

The ICFG also models an idle state, which allows the execution to wait for the next async event to occur. From every exit node in the graph, it is possible to reach the idle node. If the task queue is empty, and the last task reaches its exit node, then the control flow is interrupted (waiting in the idle-state) until an asynchronous event handler resumes the execution. This is needed for features like regular tasks implemented with timers or asynchronous processing of reads or writes, where it is a great opportunity to put the mote in an idle state, saving energy while waiting for the callback to happen.

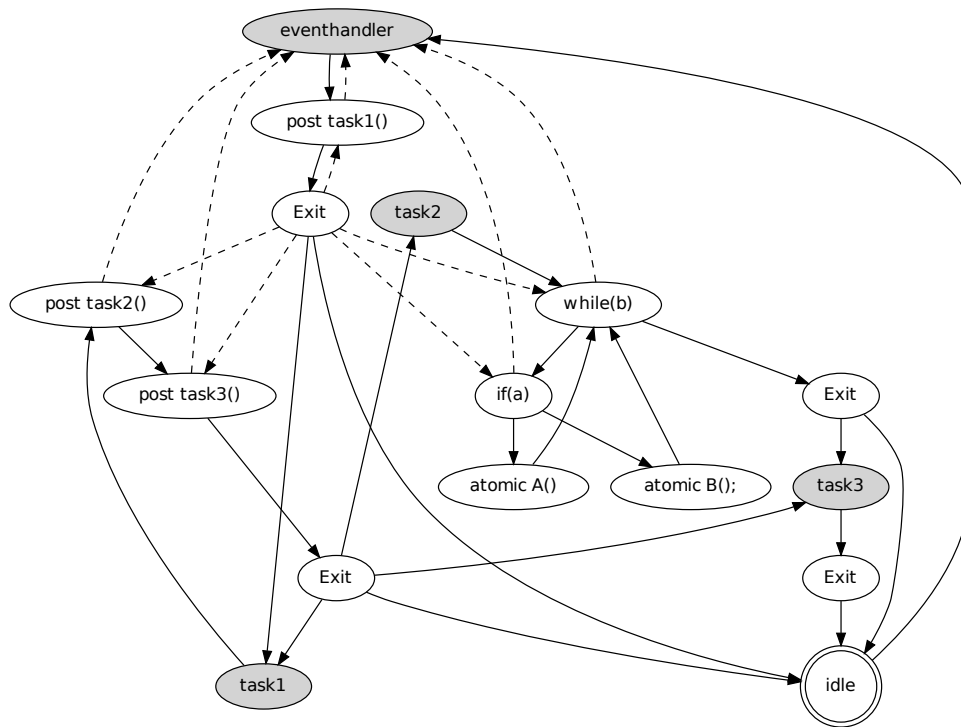


Figure 1.4.: Preemptive Execution Example



## 2. Inter-Context Control-Flow Graph

This chapter introduces the new findings. It explains the flaws of the ICFG and introduces the ICCFG which addresses these limitations.

The ICFG is, as designed, not able to cover direct calls between functions (normal function execution, building a stack). For its main target, the inter-context flow of control and data, normal, inter-context function calls do not seem important. But they may read or write data, post tasks, enable events or call commands on another context, which is not addressed by the ICFG. The construction of the Taskgraph used for the ICFG is not able to cover all transitions between tasks. The main missing transitions are those that are caused by a sequence of tasks being executed, which themselves post tasks that are added to the end of the queue. The ICFG, even with a model for normal function calls, would still not be able to allow a selection of components for the construction, because some unselected or “hidden” component might be able to transfer the control-flow from one component to another, which are both part of the construction. These transfers are lost in the ICFG.

My Work, the inter-context control-flow graph (ICCFG) is based on the inter-context flow graph developed by Lai, Cheung and Chan [5]. For an overview of its components, see the previous chapter. The ICCFG extends the ICFG with a model for normal function calls, called supergraph (see [6]), and extends the algorithms used to construct the taskgraph of Lai, Cheung and Chan to give a more complete representation of the possible transitions between tasks. The ICCFG is also able to give a precise selected component graph, because it is made possible for users to provide boundary information, which describes the hidden components on an abstract level and is used to rewrite the concerned functions according to the information given.

Intraprocedural execution is addressed with common statement-level control-flow graphs for all subroutines. Subroutine or intraprocedural execution in nesC is possible in three ways: Immediate, deferred, and preemptive. *Immediate execution* is the usual way in which functions are called and a call-stack is built. In the ICCFG, it is implemented in the spirit of the supergraph [6]. *Deferred execution* is addressed by Lai, Cheung and Chan with the taskgraph. This work extends the construction of their taskgraph to cover more of the existing paths of execution. *Preemptive execution* of interrupt handlers is taken unchanged from Lai, Cheung and Chan in the case of whole-program analysis. However, the ICCFG also supports selected component analysis. In this case, preemptive execution handling gets extended with async calls to commands and events from hidden components with the help of boundary information.

When the ICCFG should only span some of the applications components, some

of the existing control-flow traces may no longer be covered by the ICCFG directly. It is possible to recover some of the lost behavior information by manually giving boundary information for components or interfaces. This information tells which deferred subroutine calls will be made in the analyzed component by the hidden task. The ICCFG construction then rewrites the functions accordingly.

Section 2.1 gives an introduction to the model of immediate function calls with the supergraph.

Section 2.2 gives a detailed description of an example of missing transitions in the ICFG and thereby motivate Section 3.2.2.

Section 2.3 shows how the selected component analysis can gain from the boundary information given by users.

## 2.1. Modelling Immediate Execution – Supergraph

Control-flow graph nodes which contain any kind of function call need to be able to transfer the control-flow to the called function.

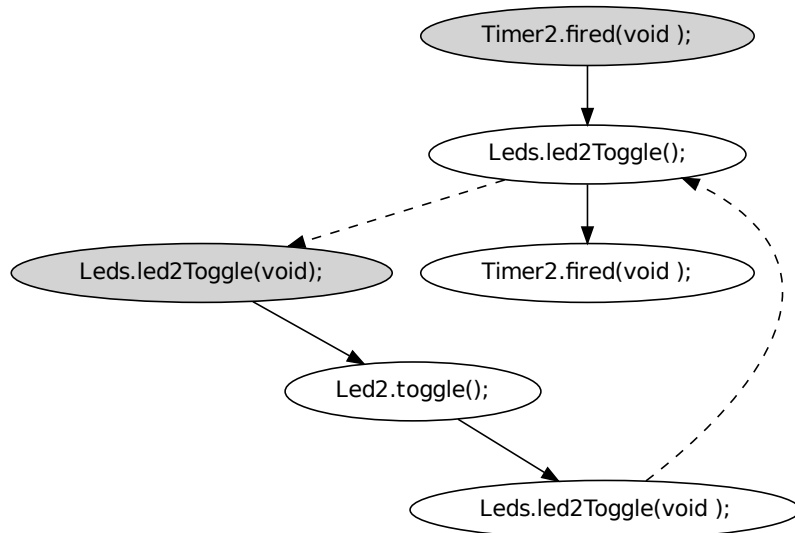


Figure 2.1.: Supergraph Example

Reps, Horwitz and Sagiv [6] split the calling node into a call and a return node and add edges from the call node to the entry node of the called function and from the exit node of the called function to the return node and build their supergraph. As the control-flow graph is processed on statement level it is not possible to split the calling node, because there may be more than one function call in a statement,

and the C language does not explicitly state in which order those function calls will be executed. This work therefore simplifies the supergraph and adds one edge from the calling node to the entry of the called function and one edge from the exit node of the called subroutine to the calling node again. This simplification is necessary because for the ICCFG a reduction of the graphs nodes to a single static assignment (or any other form, where there is a maximum of one function call per statement) was out of scope and would be very complicated, as it would have to be the same way of reduction that the destination compiler uses.

The Example in Figure 2.1 for part of the source code in Listing A.1 shows an interrupt handler `Timer2.fired` calling the `Leds.led2Toggle` function. The dashed lines show the call and the return edge to and from the called function.

## 2.2. Modelling Deferred Execution – Taskgraph

Deferred execution is already modelled by the ICFG, but their algorithms for constructing the taskgraph do not cover all possible sequences of execution.

```

module CExample
{ uses interface Boot; }
implementation
{
    task void task4 () {}
    task void task5 () {}
    task void task3 () {
        post task4 ();
        post task5 (); }
    task void task2 () {
        post task5 (); }
    task void task1 () {
        post task2 ();
        post task3 (); }
    event void Boot.booted () {
        post task1 (); }
}

```

Listing 2.1: Taskgraph Example Source

Figure 2.2 shows the task-posting trees for the functions, Figure 2.3 the taskgraph for the applications, and Figure 2.4 shows an execution trace for the Example program from Listing 2.1. On the left hand side in Figure 2.4 is the currently executed subroutine. The right hand side shows the state of the task-queue after the execution. Execution starts with the booted event, which posts its task. The other tasks are then run, according to their occurrence in the queue. When `task3` is run, the

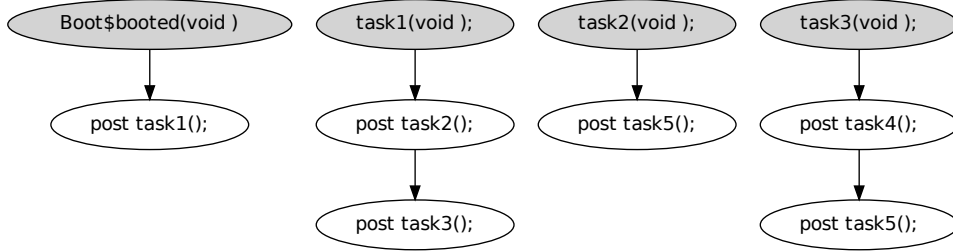


Figure 2.2.: Taskgraph Example: Task-Posting Trees

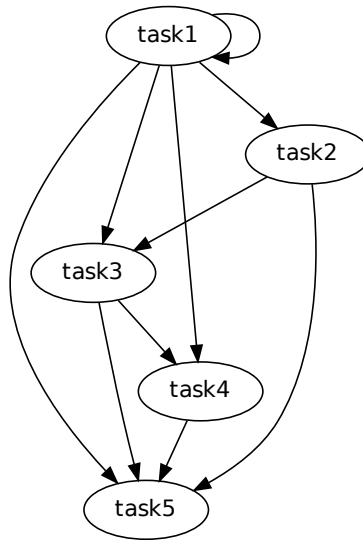


Figure 2.3.: Taskgraph Example: Taskgraph

posting of `task5` will fail for TinyOS version two, as the task is already in the queue.

This example execution trace shown in Figure 2.4 contradicts the order of tasks in the taskgraph (Figure 2.3). The execution transition from `task5` to `task4` from the trace is not part of the taskgraph. The modelling in the ICFG is not sufficient to show all execution traces. It can not show the task interdependencies which are caused by task posts of subroutines which are themselves executed deferredly.

This work proposes an additional step for processing the taskgraph, using a fixpoint iteration, which adds these missing task post. Figure 2.5 shows the fixpoint for the taskgraph from Figure 2.3. The taskgraph edges are drawn solid and the additions

```

booted: task1
task1: task2,task3
task2: task3,task5
task3: task5,task4[,task5]
task5: task4
task4:

```

Figure 2.4.: Taskgraph Example: Execution Trace

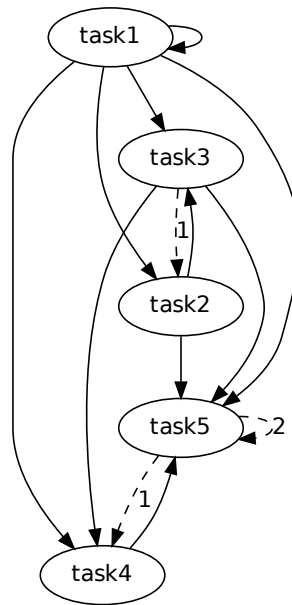


Figure 2.5.: Fixpoint for Taskgraph from Listing 2.1

are marked with the iteration number and are shown dashed. The missing edge from **task5** to **task4** is added in the first iteration, the process does not add new edges after the second iteration. The edge from **task2** to **task3** leads to the introduction of the edge from **task5** to **task4**, as the task-posting tree of **task2** ends with posting **task5** and the task-posting tree of **task3** has an entry dominated post of **task4**. It is a question whether the self-looping edges should be left out of the iteration. The edge **task1** to **task1** introduces a seeming impossible edge from **task3** to **task2** and this introduces a self-loop edge for **task5**, which is also impossible to occur.

Details of the extended taskgraph construction and processing are given in Section 3.2.2. But even this additions do not guarantee that there all possible path exist in the taskgraph. It is still neither sound, nor complete. A complete approach would

be to always assume a completely connected taskgraph.

## 2.3. Improved Split-Phase Handling

When the ICCFG covers all of the applications components, it covers all possible paths of execution. But it is not desired to always cover all of the components of the application, as this increases the size of the graph, and some analyses on it are rather expensive and a as-small as possible graph is desired.

Components left out of the ICCFG may implement some background workings and implement a split-phase interface, which is used to submit some work and then later receive results. These components are called hidden components. Simply ignoring them during the construction would loose the connection between submitting to and receiving from the components, although the actual submitting and receiving part is part of the components selected for the ICCFG, therefore my analysis needs additional information provided by the user.

The users provide a mapping between a call, specified by its component, interface, and name and a callback, specified also by its component, interface, and name. Usually those mappings are part of split-phase interfaces. It is possible to specify only the callback part, thereby allowing callbacks without a trigger call. This is used to model the `Boot.booted` callback.

The “callback” from the hidden component is possible in two basic ways: *Direct* and *indirect*. It is also possible to happen *once*, *multiple times* or *never again*. A third distinction is whether the callback is synchronous or asynchronous. The following subsections will provide details about the relevant possibilities. Modelling the *never again* callbacks could reduce the number of possible paths in the graph, if some kind of temporal modelling would be added, but this was out of the scope of this work.

All kinds of asynchronous callbacks are modelled like normal preemptive execution. *Direct* callbacks, regardless of their frequency (once or multiple) just replace their original call in the supergraph. *Indirect* callbacks are rewritten as tasks. The original call is replaced with a post to the rewritten task. If they occur multiply, the rewritten tasks task-posting tree contains a post of itself.

This work has to cover three parts of the effects in the ICCFG: The boundary information can affect the supergraph (immediate execution), the task-posting tree (deferred execution) of a subroutine and preemptive execution.

Table 2.1 shows which elements of the nesC syntax can be executed deferredly, preemptively or immediately for the whole program ( $\square$ ) and selected component ( $\diamond$ ) analysis. Tasks can only be executed deferredly by the scheduler, regardless of the selection of components. For a complete program (all component –  $\square$ ) analysis, only async events can be executed preemptively. All events, commands and helper functions can only be executed immediately forming a call stack. If the ICCFG covers only selected components ( $\diamond$ ), it is possible that some tasks or async events handlers get out of the scope. But they may signal events, or call commands in a synchronous or asynchronous manner and thereby indirectly take influence on the ICCFG.

	deferred	preemptive	immediate
task	□,◇		
async event	◇	□,◇	□,◇
event	◇		□,◇
async command	◇	◇	□,◇
command	◇		□,◇
normal function			□,◇

□: all component analysis, ◇: selected component analysis

Table 2.1.: Function Types and Execution Contexts

	direct	indirect
once	supergraph	task
multiple	supergraph	task,selfpost
never again	–	–

Table 2.2.: Rewritings for Boundary Information for Synchronous Callbacks

If the mapping points to an async callback, the callback will be rewritten as preemptively executable.

Table 2.2 shows an overview of the rewritings used for the boundary information describing synchronous callbacks. The next subsections will specify effects for direct and deferred, once and multiple synchronous callbacks and their rewritings in detail.

```

module Caller1 { ... }
implementation {
  task t1() {
    call Dummy.foo();
  }
  event Dummy.bar() {
    post t2();
    post t3();
  }
  task t2() {}
  task t3() {}
}

```

Listing 2.2: Calling Component

All of the following subsections use the same calling component, shown in Listing 2.2. It calls a command `foo` and receives an event `bar`. The event `bar` posts the tasks `t2` and `t3`.

### 2.3.1. Direct Callbacks

The hidden component might call or signal back directly as if it would have been modelled with the supergraph in two (or more) call steps.

```
module Dummy { ... }
implementation {
    command void foo() { signal bar(); }
}
```

Listing 2.3: Sync Direct Once: Example Source

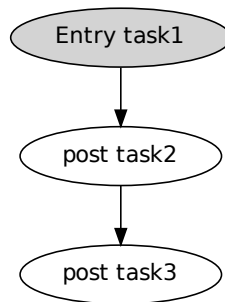


Figure 2.6.: Synchronous Direct: Task-Posting Tree Example

Listing 2.3 shows a hidden component `Dummy` and some code that calls the command `foo` from a task `t1`. The command `foo` will then directly signal the event `bar`. This could happen in a loop, signalling the event `bar` more than once, but the result will be the same on our macroscopic level.

Regardless of the count of callbacks, the command and event pair will be modelled with the supergraph. The call to `foo` will simply be replaced with a call to `bar`.

The task-posting tree will also skip the call to `foo` and inline all the task-postings done by `bar`. The resulting task-posting tree for task `t1` is shown in Figure 2.6.

### 2.3.2. Indirect Callbacks

The hidden component might use deferred execution to signal back at a later point of time *once*.

With an unhidden component the posting of task `tbar` (see Listing 2.4) would have been inlined into task `t1`'s task-posting tree and the call to `foo` would be represented in the supergraph. Task `tbar` would have called the event `bar` using the supergraph and would get `bar`'s task posts inlined in its task-posting tree.

If the boundary information describes that calling `foo` will result in a single, deferred callback of event `bar`, this behavior can be modelled by promoting the event `bar` to a task, because the hidden task would have called `bar` directly. The call to `foo` can be replaced by a post of the new task `bar`.

Figure 2.7 shows the resulting task-posting trees for Listing 2.4.

```

module Dummy { ... }
implementation {
  command foo() {
    post tbar();
  }
  task tbar() {
    signal bar();
  }
}

```

Listing 2.4: Sync Indirect Once: Example Source

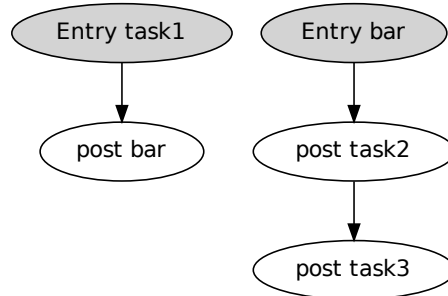


Figure 2.7.: Synchronous Indirect: Task-Posting Trees Examples

Handling *multiple* callbacks from the hidden Dummy-component (see Listing 2.5) can be modelled in nearly the same way as a single callback. The addition is that the event `bar`, which gets promoted to a task, will also be able to post itself.

The resulting task-posting trees for this example are shown in Figure 2.8.

```

module Dummy { ... }
implementation {
  command foo() {
    post tbar();
  }
  task tbar() {
    signal bar();
    post tbar();
  }
}

```

Listing 2.5: Sync Indirect Multiple: Example Source

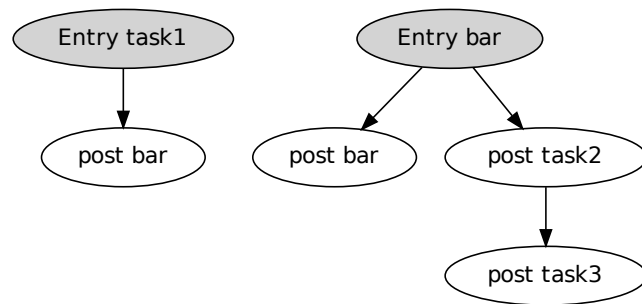


Figure 2.8.: Sync Indirect Multiple: Example Task-Posting Trees

## 2.4. The Blink Application

The Blink application is a basic example using nesC Timer components to implement led blinking. It starts three timers from its `Boot.booted` event handler during the initialization phase and then uses three event handlers (one for each Timer), to toggle its leds on and off.

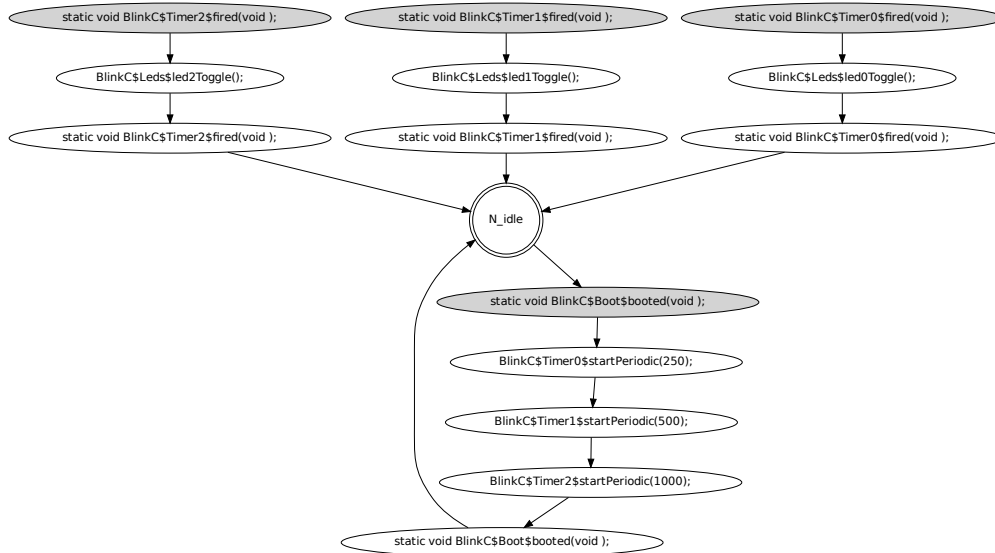


Figure 2.9.: ICFG for Blink Application

Figure 2.9 shows the ICFG for the Blink application. The entry nodes of the individual CFGs of the subroutines are marked (filled with gray). The exit nodes have a label corresponding to their entry node. It is possible to reach the `Boot.booted` event from the idle state, but all the timer event handlers seem unreachable. It is possible to stretch out the ICFG on the `LedsP` component, showing the implementation of the toggle commands, but they would also only be “tail-connected” like the timer events. They are suppressed for readability.

The ICCFG of the Blink application (Figure 2.10) shows a much more sophisticated modelling. The `Boot.booted` event is still the only subroutine reachable from the idle node, but with the help of the boundary information, it is possible, to see that any of the `fired` event handlers can be executed after the `Boot.booted` event. The ICCFG contains a fully connected taskgraph for the three `fired` events, constructed from the boundary information and the three calls to `startPeriodic`. With the help of the supergraph (shown dashed) it is possible to see that each `fired` event handler directly calls a subroutine that toggles a led for him.

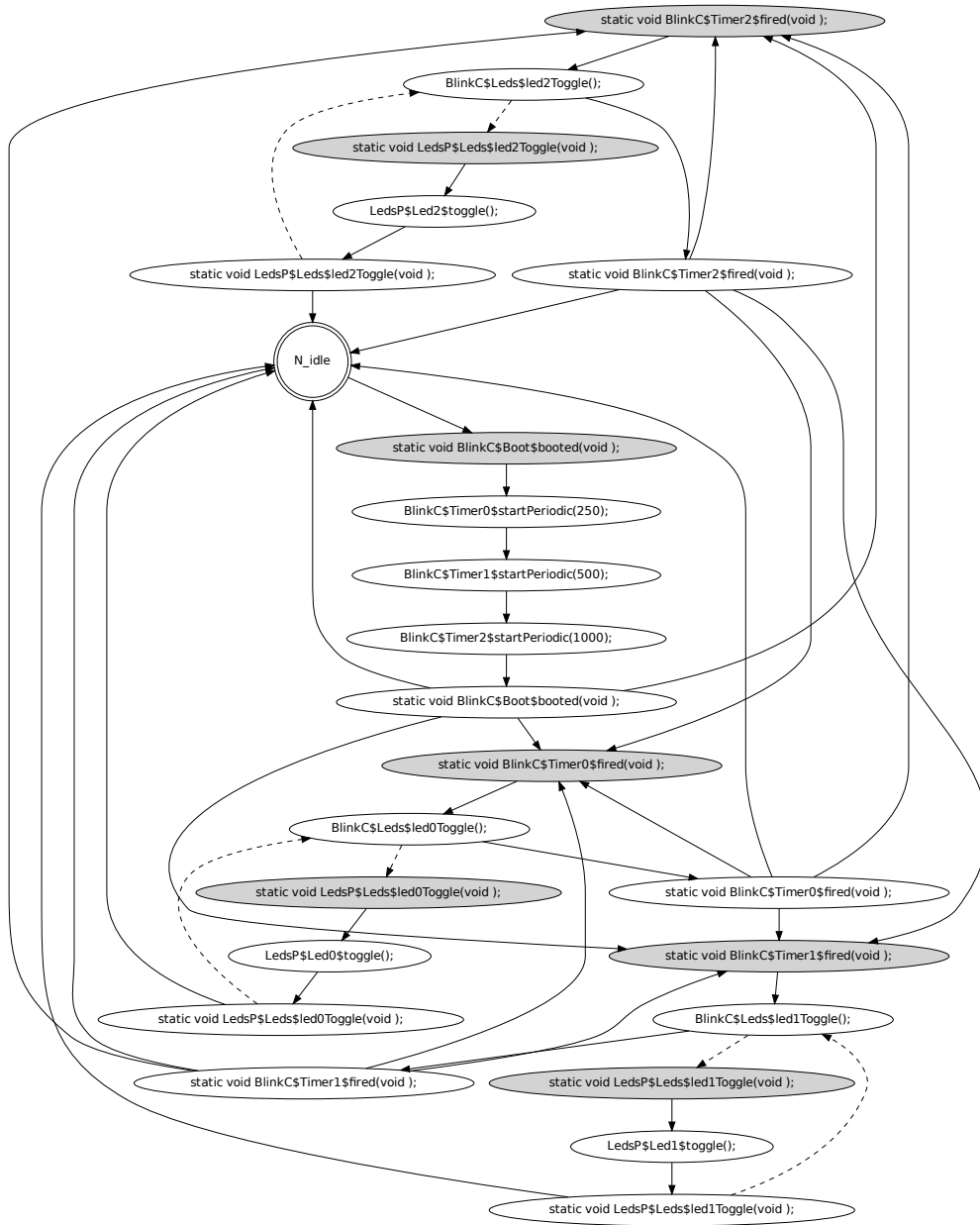


Figure 2.10.: ICCFG for Blink Application

## 3. Formal Specification

This chapter gives a formal definition of the ICCFG to allow its used as a foundation of further analyses and a precise description of its construction.

Section 3.1 gives an overview of the definitions used for the ICFG and ICCFG.

Section 3.2 describes the important algorithms used to construct the taskgraph.

### 3.1. Definition of the Inter-Context Control-Flow Graph

The definitions are subdivided as follows:

First, the basic definitions give the definition of a control-flow graph and its dominator tree. The CFGs are copied into the ICFG or ICCFG to build its foundation. The dominator tree is used to construct the task-posting trees.

Second, the task-posting trees and the taskgraph are defined. They give the foundation of the most complex construction part of the ICFG and ICCFG.

Third, the informations about different kinds of execution between subroutines are modelled with different kinds of edges that are added to the ICFG or ICCFG.

Last, the ICFG is defined and the new supergraph edges together with the ICCFG are defined.

#### 3.1.1. Basic Definitions

The basic definitions used are the following. A directed graph is a two-tuple  $(V, E)$ , containing a set of nodes  $V$  and a set of edges  $E$ . Each edge is a two-tuple  $(n_i, n_j)$  containing two nodes.

The ICCFG can represent the control flow of a complete application and also allows to cover only a selection of the applications components. An application is a set of modules  $M$ . A module  $m_k$  consist of subroutines  $u$ . The component set  $C$  is the union of the sets of subroutines of all selected modules. For all the subroutines  $u$  in the component set  $C$  control-flow graphs are constructed. The CFG is at the level of statements in the C-grammar.

**Intraprocedural Control-Flow Graph** A subroutine is represented using a directed graph  $CFG(u)$  Each control-flow graph  $CFG(u)$  has a unique start node, and a unique exit node. The other nodes of the control-flow graph represent the statements and predicates of the procedure. Edges between two nodes  $n_i$  and  $n_j$  exist, if it is possible to execute  $n_j$  directly after  $n_i$ . For edges leaving a node with a condition, attributes for the edges are used to represent the choices of the condition.

Each subroutine  $u$  with a control-flow graph  $CFG(u)$  also has a dominator tree  $DT(u)$ . A node  $n_i$  dominates node  $n_j$  in  $CFG(u)$  if every path from the entry node of  $CFG$  to  $n_j$  must go through  $n_i$ . The node  $n_i$  strictly dominates  $n_j$  if  $n_i \neq n_j$ . The immediate dominator  $n_i$  of  $n_j$  is the node, that strictly dominates  $n_j$  but does not dominate any other strict dominators of  $n_j$ .

**Dominator Tree** The dominator tree  $DT(u)$  is the tree of immediate dominators for all nodes in  $CFG(u)$ .

### 3.1.2. Task Definitions

The post statements in a subroutine define possible orderings of deferred execution of tasks. If a post statement dominates another post statement and both posts are successful, then the first posts task will be executed before the second posts task. This work uses the task-posting tree to model all possible initiations of deferred execution.

**Task-Posting Tree** A task-posting tree for a subroutine  $u$  is a directed graph  $TPT(u)$ . Each node is a post statement in  $DT(u)$ . An edge  $(n_i, n_j)$  has the following two properties: (1)  $n_i$  dominates  $n_j$  in  $DT(u)$ ; (2) if another post statement  $n_k$  ( $n_k \neq n_i$ ) dominates  $n_i$ ,  $n_k$  also dominates  $n_j$  in  $DT(u)$ .

The taskgraph gives all possible orders of deferred execution. Any trace of deferred execution is represented by a path through the taskgraph.

**Taskgraph** A taskgraph for a component-set  $C$  is a directed graph  $TG(C) = (N_C, E_C)$ , where  $N_C$  is a set of tasks in of  $C$ , and  $E_C$  is a set of directed edges representing execution orders among tasks. There is an edge  $(n_i, n_j)$  in  $E_C$  if the task  $n_j$  may be executed after  $n_i$  completes its execution.

This taskgraph is not able to benefit from the unique scheduler queue, which prohibits some paths in the taskgraph.

### 3.1.3. Interprocedural Execution

To add the taskgraph information to the ICFG or ICCFG, two kinds of task ordering edges have to be added. Edges that represent task posts of preemptively executed subroutines and edges that represent orderings of tasks from the taskgraph. *Async to task edges* are used to model the transfer the control from an async event or command to a posted task. These edges are needed for each task in  $TPT(u)$  of a preemptively executable subroutine  $u$  and its posted tasks from the exit node of  $u$  to the entry nodes of its posts. *Task to task edges* are used to model the transfer of control from one task to another. These edges are needed for each edge  $(n_i, n_j)$  in  $TG(C)$  from the exit node of  $n_i$  to the entry node of  $n_j$ .

Preemptive execution uses preemption and resumption edges to model the possible executions. Each node in a control flow graph has an attribute which marks whether

it can be interrupted by preemptive execution. A preemptively executable subroutine adds *preemption edges* from all non-atomic nodes (without entry or exit nodes) to its entry node and *resumption edges* from its exit node to all non-atomic nodes (without entry or exit nodes).

If the scheduler is in an idle state (empty queue), the execution can only be resumed by a preemptive execution. Any subroutine may be executed as the last one before the idle state is reached. *Subroutine to idle edges* model the control transfer from a subroutine to the idle state. For each subroutine  $u$  in  $C$ , add an edge from the exit node of  $u$  to an idle node. *Idle to async edges* model the control transfer from the idle state to an async event or command. For each preemptively executable subroutine  $u$ , add an edge from the idle node to the entry node of  $u$ .

### 3.1.4. ICFG

The previous definitions are put together to form the ICFG. The intraprocedural CFGs are simply copied into the ICFG and the information about interprocedural execution is added to them.

**ICFG** The ICFG of a component set  $ICFG(C)$  consists of the nodes of all of the covered control flow graphs  $CFG(u_1), CFG(u_2), \dots, CFG(u_i)$  and the idle node from the scheduler. The ICFGs edges are all of the intraprocedural edges from the covered CFGs and the *async to task*, *task to task*, *preemption*, *resumption*, *subroutine to idle*, and *idle to async* edges.

### 3.1.5. ICCFG

The ICFG definition does not cover the direct calls from one subroutine to another, forming the normal call stack. Each node in a control flow graph represents a statement which may contain immediate calls to one or more subroutines. The supergraph edges model the immediate executions in the program. An interprocedural *call to entry* edge from  $n_i$  to the start node of the called subroutine and an interprocedural *exit to return* edge from the exit node of the called subroutine to  $n_i$ .

The complete definition of the ICCFG is the ICFG extended with the supergraph edges.

**ICCFG** The ICCFG uses the same set of nodes, that the ICFG uses. The set of edges from the ICFG is extended with the *call to entry* and *exit to return* edges from the supergraph.

## 3.2. Algorithms

This section gives an abstract overview of the important algorithms used to construct the ICCFG.

The construction of the intraprocedural control flow graph of a subroutine uses a simple walker pattern, which traverses the AST and builds the CFG. Details of the implementation will be discussed in Section 4.1.

The implementation of the Lengauer-Tarjan algorithm, to construct the dominator tree from the CFGs, follows the pseudocode from [11].

### 3.2.1. Task-Posting Tree Algorithm

A task-posting tree for a subroutine is build by copying all of its dominator trees task-posting nodes into the task-posting tree.

If a subroutine contains a direct call to a task-posting function, the task-posting tree of this function should also be copied into the task-posting tree. If the subroutine is recursive, it suffices, to inline it only once, because the task posting is unique (only TinyOS version two code is processed) and any further execution will not successfully post more tasks.

For calls to commands, which have a indirect (deferred) callback given through the boundary-information, a post of the callback is inserted.

### 3.2.2. Taskgraph Construction Algorithms

The taskgraph is build from all the task-posting trees of the applications subroutines.

The taskgraph for the ICFG from [5] uses the following three algorithms: Task chains (1), tasks task posts (2) and async events task posts (4). The ICCFG uses some refined algorithms, which will also be discussed in the following with their ICFG counterparts.

#### Task Chains

Any path from the root to a leaf node in a task-posting tree is considered a task chain. A task chain is for example formed by a sequence of task posts. All chains of task-posting from the task-posting trees are copied into the task graph. If a subroutines task post is dominated by some other task post, then, if the posts succeed, this gives a possible order of task executions (see Algorithm 1). The ICFG and the ICCFG use this algorithm.

Algorithm 1 copies all parent-child relations from the task-posting tree to the task-graph.

A possible idea to shorten the chains of task posts copied into the task graph, is that each task may only come up once in a chain, because any second or further occurrence will fail anyway, because the queue will not alter its state during the execution of the subroutine which posts the chain. This idea was not processed any further because the subroutines usually do not contain such chains.

```

for all subroutine  $u \in C$  do
  for all post statement  $T_i \in TPT(u)$  do
    for all post statement  $T_j \in TPT(u)$  do
      if  $T_j$  is child node in  $TPT(u)$  of  $T_i$  then
        add  $(T_i, T_j)$  to  $E_C$ 
      end if
    end for
  end for
end for

```

Algorithm 1: TG Creation: Chains

### Task's Task Posts

The ICFG uses a simple approach to model task's task posts, described in Algorithm 2. If a task posts tasks, then it is possible that its posts can be executed immediately after its execution, if the queue was empty.

```

for all task  $T_i \in C$  do
  for all post statement  $T_j \in TPT(T_i)$  do
    add  $(T_i, T_j)$  to  $E_C$ 
  end for
end for

```

Algorithm 2: TG Creation: Tasks Task Posts (A)

Algorithm 2 adds edges from the posting task to any of its posted tasks.

This is an over approximation of the actual possibilities. If the task contains chains, they will be copied into the taskgraph by Algorithm 1. In the case of an empty queue it is only possible to continue with the head of one of the tasks chains (the task's entry dominated posts). An algorithm to implement this idea is given in Algorithm 3 and is used to construct the ICCFG. The possibility of failed posts can be discarded, because an empty queue is assumed in this case.

```

for all task  $T_i \in C$  do
  for all post statement  $T_j \in TPT(T_i)$  do
    if  $T_j$  is dominated by the EntryNode of  $TPT(T_i)$  then
      add  $(T_i, T_j)$  to  $E_C$ 
    end if
  end for
end for

```

Algorithm 3: TG Creation: Tasks Entry-Dominated Task Posts (B)

Algorithm 3 adds edges from the posting task to any of its entry-dominated task posts.

### Async Event Task Posts

Preemptive execution can happen at nearly all time of the programs execution, the tasks of preemptively executable subroutines can be executed prior to any of the other tasks or themselves. In [5] only the posts of async event handlers are considered (see Algorithm 4) to construct the ICFG, but with the selection of components it is also possible that async commands can be triggered at any time of the execution (see Algorithm 5), because they may be called by a hidden async event handler.

```
for all async event handler  $a \in C$  do  
  for all post statement  $T_i \in TPT(u)$  do  
    for all task  $T_j \in C$  do  
      add  $(T_i, T_j)$  to  $E_C$   
    end for  
  end for  
end for
```

Algorithm 4: TG Creation: Event Task Posts (A)

Algorithm 4 adds edges to the taskgraph from all of the tasks posted from async event handlers to any other task in the ICFG.

```
for all async event, async command  $a \in C$  do  
  for all post statement  $T_i \in TPT(u)$  do  
    for all task  $T_j \in C$  do  
      add  $(T_i, T_j)$  to  $E_C$   
    end for  
  end for  
end for
```

Algorithm 5: TG Creation: Preemptively Executed Subroutines Task Posts (B)

Algorithm 5 adds edges to the taskgraph from all of the tasks posted from async event handlers and async commands to any other task in the ICCFG.

### 3.2.3. Taskgraph Fixpoint Iteration

An idea to cover the fillings of a queue and new orderings of task execution not covered by analyzing single task-posting trees is to do a fixpoint iteration on the taskgraph resulting from the processing of the individual task-posting trees. If two tasks are executed directly after another, then the leaf nodes of the first task's task-posting tree represent all of its possible last task posts. The entry-dominated nodes of the next task represent its first taskposts. It is therefore possible that after each last post, any of the first posts may be executed directly.

This idea is expressed in Algorithm 6. The algorithm processes all edges given in the taskgraph and adds edges from the leaf nodes task posts of the starting point of

the original edge to the entry dominated task posts of the destination of the original edge.

```
repeat  
  for all existing edges  $n_i$  to  $n_j$  in  $TG$  do  
    add edges from all  $n_i$ 's leaf nodes to all  $n_j$ 's entry dominated nodes.  
  end for  
until  $TG$  did not change
```

Algorithm 6: TG Fixpoint Iteration

It can always be assumed, that the iteration will terminate, because every step can only add edges between some tasks and the iteration can safely be stopped, if the taskgraph does not change anymore. The worst-case fixpoint of the iteration is the fully connected taskgraph.



## 4. Constructing the ICCFG

This chapter shows what has been done. It explains how the ICCFG is constructed and describes the implementation.

The current implementation uses about 2.5 thousand lines of code.

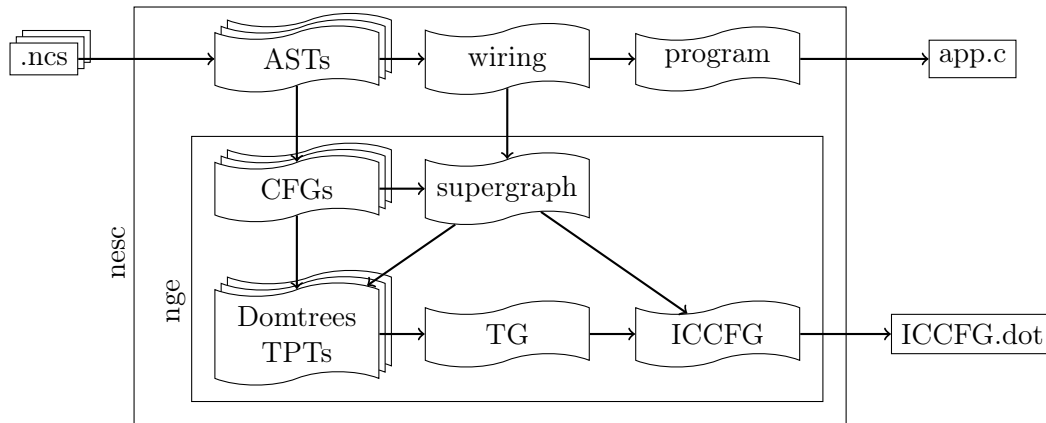


Figure 4.1.: Inner workings of the nesC Compiler

Figure 4.1 shows an overview of the process and intermediate representations used inside the nesC compiler. The starting point is given by a set of `.nc` files, that are read as necessary by the compiler and are used to build the set of ASTs. From the configuration components, the compiler extracts the wiring of the components and instantiates the used components. This wiring, together with the associated ASTs is used to build the actual program, which is unparsed into a file `app.c`. Nge, the nesC graph extractor, uses the ASTs to produce its CFGs. The wiring, together with the ASTs is used to build the supergraph. This step also addresses some of the rewritings using boundary information. The CFGs and the supergraph are then used to create dominator trees and task-posting trees (TPTs). The TPTs are then used to construct the taskgraph (TG), which is used, together with the supergraph (which incorporates the individual CFGs) to create the ICCFG. The ICCFG is traversed and the output in the dot-language is created (`ICCFG.dot`). (The use of boundary information is not show, to keep the figure readable.)

The next subsections will describe the details of this process.

Section 4.1 introduces the data structures used to keep the ASTs and CFGs and the AST walker functionality of the nesC compiler and the handlers used to construct the CFGs.

Section 4.2 describes how the boundary information is acquired from the users.

Section 4.3 describes how the CFGs are traversed to process the information needed to construct the supergraph.

Section 4.4 describes the implementation of the Lengauer-Tarjan algorithm to build the dominator tree.

Section 4.5 describes the processing of the dominator tree to form the task-posting trees.

Section 4.6 shows the implementation details of the algorithms to construct the taskgraph given on an abstract level in Section 3.2.2.

Section 4.7 the traversal of the ICCFG to form the output.

## 4.1. Converting the Abstract Syntax Tree into the Control-Flow Graph

Each nesC component gets parsed by the nesC compiler and its abstract syntax tree (AST) is saved in a data structure. This data structure contains elements which directly represent grammatical constructs.

The AST implementation in nesC uses a pseudo object hierarchy implemented by number intervals. The hierarchy is described in Lisp and then compiled to C code using emacs-lisp macros. These macros generate the type-hierarchy itself and provide additional traversal code that is used by the AST-walker, which allows an automated traversal of all the subnodes of a node. The macros also generate code for typesafe conversion between the types.

Figure 4.2 shows the AST-type hierarchy. The root node is called “node.” It uses the largest numeric interval, covering all intervals of its children. The *type\_element*, *expression* and *declaration* parts are not used for the ICCFG and are shown collapsed. The statement type and its children are shown completely.

This implementation processes the AST using a walker routine, which is provided by the nesC compiler infrastructure. It allows one to register handlers for certain grammatical constructs and traverses the complete AST. An outer walker is used to locate all function definitions of a component. These also include task and event definitions.

If a function definition is found, a basic datastructure to hold the information of the control-flow graph (CFG) is created. A second (inner) walker is started to process the subtree of the function definition. Its handlers are listed in Table 4.1. The table shows the statement type for which the handler is registered, which nodes representing statements are generated, which meta-nodes (without associated statement) are generated, which walker state is written to, and which walker state is used by the handler.

The state is used to let the construction of the CFG nodes know, in which context they exist. `in_atomic` is used to signal, whether the current execution is contained in an `atomic` statement or block and should therefore be unpreemptable. `continue`, `break`, and `exit` are used to allow control-flow statements like `return`, `break`, or

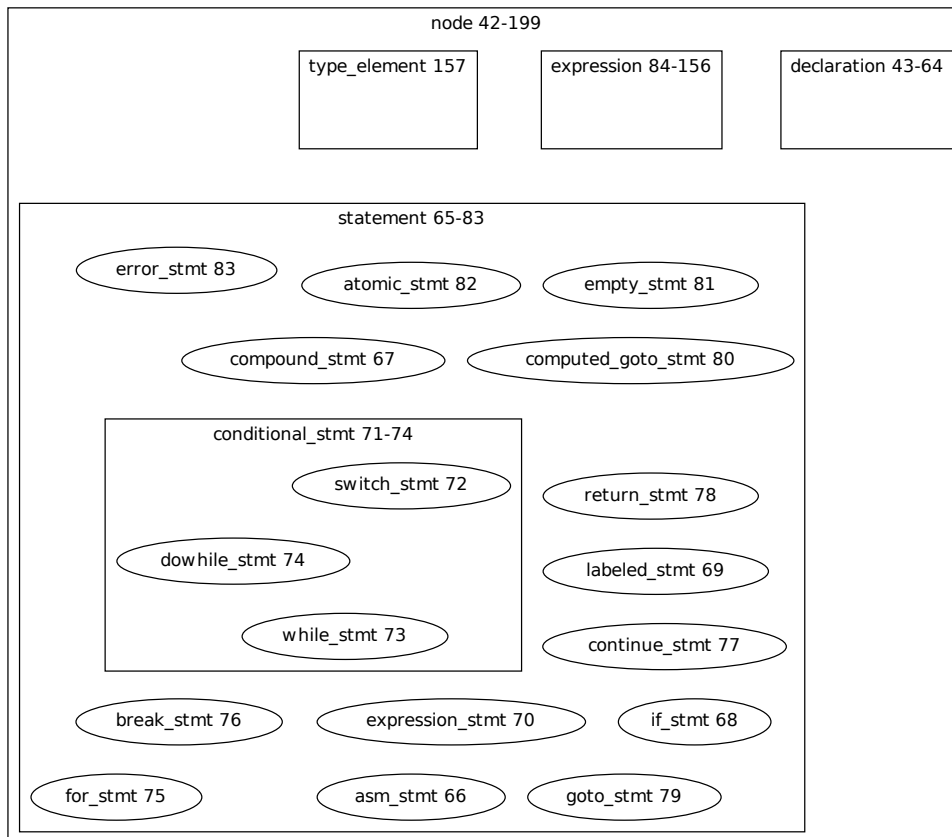


Figure 4.2.: AST Types used by the NesC Compiler

`continue` to find their jump destinations. The `labels` state is used to collect the destinations of jumps of the `switch` or `goto` constructs.

Each handler constructs nodes for the control-flow graph representing the statements that are contained in its subtree. Some walkers need to create nodes which are not associated with a statement. These are mostly for convenience reasons during the construction of the CFG and are later cleaned from the CFG.

The atomic handler alters the state of all its subtrees walkers to mark their created nodes as atomic. The `while`, `dowhile` and `for` handlers process the explicit looping constructs of the nesC language. They need to set the break and continue state of their subtree-walkers to allow the lookup of `break` and `continue` statements destinations. The `break`, `continue`, `return` and `goto` handlers form a group of handlers which allow for direct transfer of the control flow. They all look up their destinations in the walker state. The labeled statement handler processes the `goto` and `switch` labels of statements and sets the walker state, allowing the previous group of handlers

Type	Nodes	Meta Nodes	State Written	State Used
atomic		pre, post	in_atomic	
while	cond	after	continue, break	in_atomic
dowhile	cond	do,post	continue, break	in_atomic
for	init, cond, post	after	continue, break	in_atomic
break	break			in_atomic,break
continue	continue			in_atomic,continue
return	return			in_atomic,exit
goto	goto		labels	in_atomic,labels
labeled	label		labels	in_atomic,labels
if	cond	after		in_atomic
switch	cond	post	labels,break	in_atomic

Table 4.1.: AST Walker Handlers

to discover their destinations. The last group of handlers manages the selections of control flow using a predicate (`if`) or expression (`switch`). The `switch` handler needs to inter-operate with the labeled statement handler to detect its destinations.

Each node that represents a statement processes its subtree with a third walker, which detects function calls (also event signal and task post) in its subtree.

A node in the CFG is represented with a struct of type `ngeblock` (see Listing 4.1). Its important members are: `defaultnext` and `valuednext`, representing the control-flow and `specials`, the list of function calls, event signals, task posts and command calls of a nodes statment. `defaultnext` is a pointer to another `ngeblock`. It is always filled, as each node (except the exit node) has a next node in the control flow. `valuednext` is a list of structs containing pointers to values of type `expression` and a pointer to a `ngeblock`. For nodes with predicates, the `valuednext` list contains the possible evaluations of the predicate, which alter the control-flow.

## 4.2. Boundary Information

This work allows the user to provide a helper file with boundary information which gives causal mappings for components. If the boundary information only specifies the callback part of the mapping, it is assumed that this callback will be triggered regardless of any precondition.

Boundary information is read from a helper file. Figure 4.3 shows its grammar. It gives a mapping from and endpoint to an endpoint, both represented by a 3 tuple consisting of component, interface, and function names. There is no syntactic distinction between commands and events, as this would only limit further use. The first tuple usually represents a command, the second an event. Each mapping has a parameter that specifies the frequency of the callback with the weight parameter ( $-1$  is multiple,  $1$  is once,  $0$  is never again) and the type parameter ( $0$  is direct,  $1$  is indirect).

```

typedef struct nextblock {
    struct ngeblock *block;
    expression value;

    struct nextblock *next;
} nextblock;

typedef struct specialblock {
    nesc_call_kind kind;

    node node; // source fkt call node

    endp source, destination;
    struct nge_cfg *dest_cfg;

    struct specialblock * next;
} *specialblock;

typedef struct ngeblock {
    // pointer to next blocks
    nextblock valuednext;
    struct ngeblock *defaultnext;

    // pointer to specials within the block
    specialblock specials;
} *ngeblock;

```

Listing 4.1: (Partial) Structs for CFG Nodes

### 4.3. Supergraph

All CFGs generated for the subroutines are processed and if they contain any kind of function call that is not a task post, the boundary information is checked, whether they should be rewritten. This step also looks up the direct calls, that are modeled in the supergraph.

The supergraph does not need any further processing and is not saved explicitly, its edges are emitted during the traversal of the ICCFG (see Section 4.7). All the function calls can find their destinations without complications due to the use of canonic names for all the nodes in the graph.

```

endpoint:
    component-name interface-name function-name
weight: one of
    -1
    0
    1
type: one of
    0
    1
call:
    endpoint
callback:
    endpoint
parameter:
    [weight= weight ,type= type ]\n
mapping: one of
    -> callback parameter
    call -> callback parameter

```

Figure 4.3.: Boundary Information Grammar

## 4.4. Generating the Dominator Tree for a Control Flow Graph

The Lengauer-Tarjan algorithm gives an almost linear solution to the problem of finding immediate dominators. For this performance it needs a fast data structure to handle the sets it operates on. To simplify the implementation, the hash-table data structures provided by the nesC compiler are reused. Their lookup implementation is not fast enough (only linear) to allow the Lengauer-Tarjan algorithm to get to its full speed. This is not a problem, because the dominator trees calculated only cover intraprocedural control-flow graphs, which tend to be not very long.

The implementation follows the pseudo-code implementation of the algorithm given in [11, pp. 452, 453].

This implementation saves the dominator tree in two arrays, one containing the nodes of the tree, and the second the immediate dominators for each of the nodes.

## 4.5. Creating a Task-Posting Tree from a Dominator Tree

The task-posting tree is created by traversing the dominator tree. As it is saved in two arrays, a traversal starting in the root node needs to look-up each child of the root node by comparing all of the node's immediate dominator entries. If such a

node contains any kind of function call, it is possible a part of the task-posting tree. Task posts and calls rewritten to task posts using boundary information get added directly. Any other type of call (direct function calls to the original or rewritten destination), should get their task-posting trees inlined here. The inlining is only possible in a second traversal, because in the first, not all of them are created. A recursive inlining does not seem promising, as the task posts are unique for TinyOS version two, and a second traversal of the TPT would not change the tasks posted.

The inlining of TPTs is currently not implemented. This is a major drawback, because applications are known to use helper functions, that are directly called to post tasks for them [2]. Its implementation was put to hold until the problem with task-post rewriting by the nesC compiler is solved. The nesC compiler, when used in the normal TinyOS toolchain, gets a scheduler parameter, which defines the scheduler component to be used and the rewritings for the tasks and task posts. All task posts are rewritten to command-calls on the scheduler component and all tasks are rewritten as events, that are signaled by the scheduler. Omitting the scheduler parameter introduces a huge amount of problems with the syntactical checking of the nesC compiler, which apparently cannot cope with tasks. These two problems are to be addressed in a next version of the tool.

The limitations by these two problems still allow the generation of taskgraphs in a “lab” environment, by directly calling the nesC compiler with a reduced parameter set and a reduced application sourcecode. The taskgraph for Section 2.2 was generated in this way.

For simplification in the implementation, the task-posting tree can be implemented as DAG. This allows a simplified inlining of other task-posting trees, because the subtrees below the inlining do not have to be duplicated. The nature of a DAG allows a very simple conversion to a tree later.

The task-posting tree and taskgraph share the same data structure `nge_taskgraph`. It contains a `next` pointer to items on the same level, and a `children` pointer to descend one level. All of one nodes children are reachable by first traversing the `children` pointer once and then iterating over the `next` pointers.

## 4.6. Processing the Task-Posting Trees to Form a Taskgraph

Because the task-posting trees are processed by different algorithms, the implementation provides a traversal function, which takes a function pointer as one of its arguments. The parameterization allows one to easily implement many processings of the task-posting trees.

The taskgraph distinguishes tasks simply by their name. The taskgraph is preseeded with all the available tasks (giving its set of nodes) and edges are only added if they are not already added.

The taskgraph iteration as described in Section 3.2.3 is currently not implemented but it will use the same data structures. This omission does not limit the practical

scope of the tool until the automatic rewriting of tasks done by the scheduler problem is solved. But even in this case, TinyOS applications usually do not use very complex task posting models, that go far beyond posting task from event handlers.

## 4.7. Traversing the ICCFG

The ICCFG is generated in the dot format. It allows easy further processing by other tools and graphviz provides tools to visualize graphs using the dot format.

The implementation traverses all CFGs for the subroutines. It generates canonic names for the nodes consisting of a prefix of `component`, `interface` and `function` separated by underscore, if the subroutine is part of a component's outer interfaces, or the function name if it is not. Entry nodes are suffixed with `_nge_start`, exit nodes use `_nge_stop`. Any other node uses the number it was assigned during a depth-first-search, within the suffix `_nge_%i`, replacing `%i`.

The output also generates labels for the nodes representing statements by processing their AST elements with the unparser of the nesC compiler. This does not restore the original nesC code, it can only output the destination C code. Because the labels are only for human readability this drawback was acceptable so far. Restoring the nesC code would require the implementation of unparsers for most of the grammar of nesC.

The implementation of the output of the taskgraph, task-posting trees, and idle edges is straightforward by traversing the respective elements and writing edges to the output.

The supergraph, preemption and resumption edges are processed during the traversals of the individual CFGs, adding edges to and from the called functions.

All edges types are printed using a `mytype` attribute, stating their type. This allows an easy evaluation of the generated ICCFG.

## 5. Results

This chapter gives a resume of the work. It shows the tests of the ICCFG-tool and their results and describes ideas for future work.

### 5.1. The Testset

The TinyOS version two distribution comes with a directory of demonstration applications. This demo application directory was chosen as a testset to check the generation of the ICCFGs. The directory contains of several subdirectories, some of them containing TinyOS applications written in nesC. Table 5.1 shows that there are 116 subdirectories in total, with 24 not containing nesC applications. The compilation of the applications was tested with the generation of the ICCFG turned on and of. 21 of the 92 directories with applications were not compilable for the `micaz` or `iris` platform. The remaining directories were compiled with the components corresponding to their top-level `.nc` selected as components.

For 67 applications ICCFGs were generated. For one application the tool warned about an empty CFG (`tests/TestFtsp/FtspLplBeaconer`), although the top-level component contained source-code. One application (`tosthreads/apps/Blink`) caused the tool to hit the timeout of five seconds used for the compilation. Two applications contained only wiring and no control-flow information (`tests/deluge/Basestation` and `tests/deluge/GoldenImage`).

The table for the ICCFGs from the demonstration directory will give an overview of the edge and node counts for each ICCFG and discuss possible findings.

116	subdirectories containing a <code>Makefile</code>
24	subdirectories without <code>.nc</code> files in the top directory
92	to be checked
21	uncompilable a priori
2	only with wiring
1	tool timed out
1	without ICCFG generated
67	with ICCFG generated

Table 5.1.: Contents of the TinyOS Demo Application Directory

Application	C	LOC	nodes	ipcfg	tpt	tg	idle	presum	sg
AntiTheft/Nodes	2	295	95	92	0	0	31	0	12
AntiTheft/Root	2	173	36	34	0	0	17	0	0
BaseStation15.4	2	283	73	72	0	0	22	0	5
BaseStation	2	390	135	144	0	0	28	0	8
Blink	2	132	17	13	3	15	6	0	0
MultihopOscilloscope	2	351	127	137	0	0	30	0	20
MultihopOscilloscopeLqi	2	353	129	139	0	0	30	0	22
Null	2	99	2	1	0	0	3	0	0
Oscilloscope	2	176	62	62	0	0	19	0	9
Powerup	2	95	3	2	0	0	3	0	0
Sense	2	143	18	19	0	0	7	0	0
tests/arbiters/TestFcfsArbiter	2	204	29	22	0	0	15	0	0
tests/arbiters/TestRoundRobinArbiter	2	204	29	22	0	0	15	0	0
tests/cc2420/TestAcks	2	176	30	26	0	0	15	0	0
tests/cc2420/TestSecurity/BaseStation	2	399	138	146	0	0	30	0	8
tests/cc2420/TestSecurity/RadioCountToLeds1	2	222	49	53	0	0	15	0	0
tests/deluge/Blink	2	119	7	5	0	0	5	0	0
tests/deluge/SerialBlink	2	128	11	8	0	0	7	0	0
tests/storage/Block	2	265	130	130	0	0	31	0	33
tests/storage/CircularLog	2	266	130	133	0	0	30	0	31
tests/storage/Config	2	316	153	156	0	0	31	0	36
tests/storage/Log	2	288	145	152	0	0	31	0	37
tests/storage/SyncLog	2	148	56	53	0	0	25	0	0
tests/TestAdc	2	168	26	23	0	0	13	28	0
tests/TestDissemination	2	212	37	37	0	0	13	0	0
tests/TestEui	2	106	16	14	0	0	6	0	0
tests/TestFtsp/FtspLpl	2	186	41	39	0	0	13	0	0
tests/TestLpl	2	206	54	57	0	0	14	0	1

Application	C	LOC	nodes	ipcfg	tpt	tg	idle	presum	sg
tests/TestNetwork	2	289	109	104	0	0	32	0	2
tests/TestNetworkLpl	2	282	105	99	0	0	32	0	2
tests/tkn154/beacon-enabled/TestAssociate/coordinator	2	202	38	29	0	0	25	0	0
tests/tkn154/beacon-enabled/TestAssociate/device	2	296	59	55	0	0	24	0	3
tests/tkn154/beacon-enabled/TestData/coordinator	2	165	32	26	0	0	19	0	0
tests/tkn154/beacon-enabled/TestData/device	2	273	58	59	0	0	18	0	2
tests/tkn154/beacon-enabled/TestIndirect/coordinator	2	201	41	35	0	0	20	0	2
tests/tkn154/beacon-enabled/TestIndirect/device	2	216	37	33	0	0	16	0	2
tests/tkn154/beacon-enabled/TestStartSync/coordinator	2	150	25	21	0	0	15	0	0
tests/tkn154/beacon-enabled/TestStartSync/device	2	220	46	48	0	0	12	0	4
tests/tkn154/nonbeacon-enabled/TestActiveScan/coordinator	2	144	26	19	0	0	17	0	0
tests/tkn154/nonbeacon-enabled/TestActiveScan/device	2	184	33	30	0	0	15	0	0
tests/tkn154/nonbeacon-enabled/TestAssociate/coordinator	2	188	29	23	0	0	17	0	0
tests/tkn154/nonbeacon-enabled/TestAssociate/device	2	211	44	39	0	0	18	0	2
tests/tkn154/nonbeacon-enabled/TestIndirectData/coordinator	2	209	45	40	0	0	18	0	3
tests/tkn154/nonbeacon-enabled/TestIndirectData/device	2	162	31	24	0	0	16	0	1
tests/tkn154/nonbeacon-enabled/TestPromiscuous	2	210	84	91	0	0	13	0	0
tosthreads/apps/BaseStation	3	416	8	6	0	0	5	0	0
tosthreads/apps/Blink_DynamicThreads	2	118	15	17	0	0	4	0	0
tosthreads/apps/Bounce	2	243	32	34	0	0	11	0	0
tosthreads/apps/RadioStress	2	196	26	28	0	0	9	0	0
tosthreads/apps/TestBasicsbSensors	2	148	15	18	0	0	5	0	0
tosthreads/apps/TestBlockStorage	2	159	26	29	0	0	5	0	0
tosthreads/apps/TestCollection	2	230	47	50	0	0	9	0	4
tosthreads/apps/TestJoin	2	157	23	22	0	0	9	0	0
tosthreads/apps/TestPrintf	2	114	5	4	0	0	5	0	0
tosthreads/apps/TestSineSensor	2	137	12	14	0	0	5	0	0
tutorials/BlinkConfig	2	192	37	41	0	0	13	0	0

Application	C	LOC	nodes	ipcfg	tpt	tg	idle	presum	sg
tutorials/BlinkFail	2	137	18	14	0	0	9	0	0
tutorials/BlinkTask	2	120	9	6	0	0	7	0	0
tutorials/BlinkToRadio	2	180	40	42	0	0	14	0	1
tutorials/LowPowerSensing/Base	2	204	42	36	0	0	19	0	0
tutorials/LowPowerSensing/Sampler	2	198	48	45	0	0	19	0	5
tutorials/PacketParrot	2	214	50	48	0	0	23	0	0
tutorials/Printf	2	102	7	6	0	0	3	0	0
tutorials/RssiDemo/InterceptBase	2	410	141	150	0	0	32	0	8
tutorials/RssiDemo/RssiBase	2	142	7	5	0	0	4	0	1
tutorials/RssiDemo/SendingMote	2	115	13	8	0	0	11	0	0
tutorials/SharedResourceDemo	7	483	76	59	0	0	35	0	2
<i>Averages</i>	2.09	210	49.5	48.5	0.0448	0.224	16.2	0.418	3.97

Table 5.2.: Automated Evaluation of ICCFGs from the Demo Application Directory

## 5.2. ICCFGs for the Demo Applications

The ICCFGs for the demo applications were generated using the new compiler option `-fnesc-nge-iccfg` to trigger ICCFG generation. The components provided as `.nc` files for each application in their directory were used as the selected components, using the new `-fnesc-nge-components=...` comma-separated list option for component selection.

Table 5.2 shows the results of the automated generation and evaluation of the demo applications. The table rows represent the following: `C` is the number of components selected for the ICCFG; `LOC` is the number of lines of code for the selected components; `nodes`, the number of nodes in the ICCFG; `ipcfg`, the number of intraprocedural control-flow edges; `tpt`, the number of task-posting tree edges; `tg`, the number of edges in the taskgraph; `idle`, the number of edge leading to and from the idle state; `presum`, the number of preemption and resumption edges and `sg`, the number of supergraph edges. The averages are shown with a precision of 3 decimal digits.

Most of the analyzed applications have two `.nc` files in their top-level directory and therefore two components were selected for the ICCFGs. The number of lines of code for the applications ranges between 99 and 483 with an average of 210 lines of code. The ICCFGs contained between two and 153 nodes with an average of 49.5 nodes. There were between one and 156 (average 48.5) intraprocedural edges in the ICCFGs. Taskgraph and task-posting tree edges were only seen for the `blink` application. The ICCFGs between three and 35 (average 16.2) edges to and from the idle node. The only application to contain preemption and resumption edges was the `TestAdc` application. The number of supergraph edges ranged between zero and 37 with an average of 3.97.

## 5.3. Interpretation

Some entries in the Table 5.2 deserve an explanation. For nearly all automatically tested components, the number of components was two. This represents the common TinyOS pattern of providing a wiring top-level component and an application-code component in the application directory and using system components for implementing low-level details.

The number of lines of code is in the range from 99 to 483, showing how well the TinyOS abstractions work. The actual applications do not need to contain lots of code, with most of the code hidden in the system components. The `Null` and `Powerup` applications mark the lower end of the range, as they are minimal implementation applications, just allowing compilation without any real application code.

The application `TestAdc` is the only application with preemption and resumption edges, because it is the only one implementing async event handlers in the top level.

The ICCFGs showing supergraph edges, which mostly represent inner component calls to helper functions, mark applications with more implementation in the top-level components.

The Blink application used boundary information to process its hidden Timer component that produced a task-posting tree and a taskgraph.

### 5.3.1. Implementation

The numbers show that the ICCFG-tool can generate intraprocedural control-flow graphs, generate the supergraph, process async event handlers and edges to the idle node for all the applications in the demo directory. Selected component analysis is possible and when it is used with boundary information leads to the generation of task-posting trees and a taskgraph.

Due to a Bug in the ICCFG-tool it was not possible to process the task and task-posting parts of the applications themselves, because in the real-world compilation process used, the nesC compiler replaces all task posts with calls to an async command implemented by the scheduler and all tasks with synchronous event handlers. It is possible to suppress this replacement, but then the compiler complains massively about broken module interfaces. It is necessary to use the real-world compilation process, because the applications make use of a lot of TinyOS components and the compiler would complain about their missing, if called with a reduced set of compilation flags. (It is still possible to generate the taskgraph of an ICCFG automatically. Figure 2.3 shows an automatically generated taskgraph from the Listing 2.1.)

The inlining of task-posting trees which is supposed to cover task-posting of helper functions that are directly called is not implemented. Also the taskgraph-iteration is not implemented. There is no consequence of the missing implementations, because it is not possible to generate the task-posting trees for real world applications due to the scheduler rewriting problem.

## 6. Conclusion and Future Work

**Conclusion** The ICCFG provides improvements over the ICFG as it is now possible to model direct calls from one function to another. Moreover, it provides a fixpoint iteration for the taskgraph, which solves the problem of tasks task posts for different orderings of tasks and it can use boundary information provided by the user, to allow the ICCFG to cover only selected components and yet capture this interaction with the unselected ones.

There are still drawbacks in the ICCFG. It does not benefit much from the uniqueness of the task queue of TinyOS version two, which eliminates some traces of execution in the taskgraph, because the ICCFG does not use the necessary temporal modelling. Also the taskgraph itself is neither proven sound nor proven complete. Another approach for taskgraph construction would be to assume all possible execution orders of tasks, which would be represented by a fully connected taskgraph. One could then exclude some known-impossible edges from the taskgraph, leading to a safe assumption.

**Future Work** Future work could concern the temporal relations of the ICCFG, namely that events can be enabled and disabled at a later point in time (for example a Timer being started, firing some times and then being stopped again; or a callback only returning once) and limiting the possible control-flow in the application.

Along with one could temporal modelling introduce symbolic execution, which would further reduce the possible control flow, for example by eliminating unreachable code or by distinguishing different states of the application whenever only a part of the complete control flow is possible.

This would lead directly to the modelling of state machines representing the applications on an abstract level.

The future implementations of the ICCFG could use the nesC compiler infrastructure just for the extraction of the AST and wiring of the components and then process the extracted data in a programming language that gives more support to process graphs. This would open the generation of the ICCFG to a broader field of users, as it would be easier to manipulate the ICCFG or start a new analysis on top of it.



# A. Appendix

## A.1. Blink Application Source

```
#include "Timer.h"
module BlinkC @safe()
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }
  event void Timer0.fired()
  {
    dbg("BlinkC", "Timer_0_fired_@_%.2s_\n", sim_time_string());
    call Leds.led0Toggle();
  }
  event void Timer1.fired()
  {
    dbg("BlinkC", "Timer_1_fired_@_%.2s_\n", sim_time_string());
    call Leds.led1Toggle();
  }
  event void Timer2.fired()
  {
    dbg("BlinkC", "Timer_2_fired_@_%.2s_\n", sim_time_string());
    call Leds.led2Toggle();
  }
}
```

Listing A.1: Source Code for Blink Application

## A.2. RegionC

The nesC compiler is implemented in RegionC. RegionC is a dialect of C, implementing a manual region-oriented memory management strategy. Every allocation is contained in a region, memory is freed by deleting a region and regions may be built hierarchically.

RegionC provides safe deletion of memory regions by implementing reference counting to memory regions. RegionC uses its own compiler RC. It is compatible to normal C compilers, because an implementation for allocating and deleting regions is given by a header file. The header does not implement the reference counting mechanisms.

RegionC was developed by David Gay and its codebase is an ancestor of the nesC codebase [12].

# Bibliography

- [1] S. Georgi, C. Weyer, M. Stemick, C. Renner, F. Hackbarth, U. Pilz, J. Eichmann, T. Pilsak, H. Sauff, L. Torres, K. Dembowski, and F. Wagner, “Somsed: An interdisciplinary approach for developing wireless sensor networks,” in *7. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*, 2008, pp. 08–12.
- [2] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon, “Health monitoring of civil infrastructures using wireless sensor networks,” in *In IPSN '07: Proceedings of the 6th International Conference on Information Processing in Sensor Networks*. ACM Press, 2007, pp. 254–263.
- [3] K. Dembowski and B. T. Bradford, “Low power wireless sensor node,” *Proceedings Self-Organizing Mobile Sensor and Data Networks (SOMSED) Workshop*, 2009.
- [4] S. Roundy, “On the effectiveness of vibration-based energy harvesting,” *Journal of Intelligent Material Systems and Structures*, vol. 16, no. 10, pp. 809–823, October 2005. [Online]. Available: <http://dx.doi.org/10.1177/1045389X05054042>
- [5] Z. Lai, S. C. Cheung, and W. K. Chan, “Inter-context control-flow and data-flow test adequacy criteria for nesC applications,” in *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2008, pp. 94–104.
- [6] T. Reps, S. Horwitz, and M. Sagiv, “Precise interprocedural dataflow analysis via graph reachability,” in *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1995, pp. 49–61.
- [7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, “The nesC language: A holistic approach to networked embedded systems,” in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2003, pp. 1–11.
- [8] B. W. Kernighan and D. M. Ritchie, *The C Programming Language, Second Edition*. Englewood Cliffs, New Jersey: Prentice-Hall, 1988.
- [9] D. Gay, P. Levis, D. Culler, and E. Brewer, *nesC 1.3 Language Reference Manual*, 2009.

- [10] P. Levis and C. Sharp, “TinyOS extension proposal (tep) 106: Schedulers and tasks,” 2006.
- [11] A. W. Appel, *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.
- [12] D. Gay and A. Aiken, “Language support for regions,” in *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2001, pp. 70–80.
- [13] R. Jhala and R. Majumdar, “Interprocedural analysis of asynchronous programs,” in *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2007, pp. 339–350.
- [14] N. Kothari, T. Millstein, and R. Govindan, “Deriving state machines from TinyOS programs using symbolic execution,” in *IPSN '08: Proceedings of the 7th International Conference on Information Processing in Sensor Networks*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 271–282.
- [15] V. Menrad, M. Garcia, and S. Schupp, “Improving TinyOS developer productivity with statecharts,” *Proceedings Self-Organizing Mobile Sensor and Data Networks (SOMSED) Workshop*, 2009.