Technische Universität Hamburg-Harburg

Diplomarbeit

An Interprocedural Points-To Analysis for Event-Driven Programs

Timo Kamph

January 2, 2012

supervised by:
Prof. Dr. Sibylle Schupp
Prof. Dr.-Ing. Andreas Timm-Giel

Technische Universität Hamburg-Harburg
Institute for Software Systems
Schwarzenbergstraße 95
21073 Hamburg

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, 2. Januar 2012

Timo Kamph

Contents

List of Figures

1 Introduction

The analysis and verification of programs is difficult if the programming language supports pointers. Without any knowledge about the targets of a pointer, one must conservatively assume that any pointer dereference may yield access to any variable in the program. In general, this leads to a significant loss of precision in the statements one can make about a program. Therefore, pointer analyses have been devised that compute a set of possible targets for a pointer and can substantially reduce the number of variables that must be assumed to be accessed through that pointer. Most of the existing analyses are for sequential programs and do not support asynchronous, event-driven programs, where the non-deterministic execution order of events complicates matters. This thesis provides an interprocedural, flow-, and context-sensitive points-to analysis for event-driven programs, to foster the analysis and verification of these programs.

The goal of a points-to analysis is to identify all possible targets of a pointer. Since this problem is, in general, undecidable, points-to analysis algorithms compute a safe approximation of the actual points-to relations. The precision of such an analysis is determined by the number of wrongfully assumed points-to relations and depends on the properties of the analysis. To achieve a high precision, the presented analysis is flow-sensitive, that is, it considers the control-flow and determines the points-to relations for every program point individually. Furthermore, it is an interprocedural analysis that considers the effect of a called function on the points-to relations. This evaluation of function calls is context-sensitive, i.e., a function is analyzed for every call site separately, with exactly the points-to relations that were computed for that particular call site.

The control flow of event-driven programs is influenced by the occurrence of external events. Whenever an event is signaled to the program, the corresponding event handler is invoked. In general, the occurrence of events is non-deterministic, and therefore the execution order of event handlers is also non-deterministic. This is a challenge for program analyses, as they must consider all possible execution orders to be sound, while including unrealizable execution orders makes the result imprecise. The existing analyses for sequential programs do not account for non-deterministic control flow, and are therefore unsuited for event-driven programs. The points-to analysis presented in this thesis is based upon a points-to analysis for multithreaded programs and models all possible execution orders of the program's events.

The considered input language for the analysis algorithm is a subset of nesC, a programming language for component-based, event-driven applications. It includes assignment statements, control-flow statements, function calls, and events. Instead of working on the input program directly, the analysis uses an intermediate program representation: control-flow graphs for the intraprocedural analysis and an event graph for the event-sensitive analysis. The event graph is an adaptation of the parallel flow graph for multithreaded programs and a novel approach to model the non-deterministic execution order of events.

The presented analysis is implemented as a prototype in the nesC compiler. The algorithm and the prototype were evaluated by analyzing some programs. For simple programs that adhere to the defined input language, correct points-to relations were obtained. More complex, real world nesC programs require some extensions to the analysis and implementation. The important features that are missing are support for typecasts in the input language, handling of parameters and return values in the implementation, and an extension of the analysis beyond the top-level module of an application.

The remainder of this thesis is structured as follows: Chapter 2 provides an overview of data-flow analyses in general, and pointer analyses in particular. Additionally, nesC is introduced as a programming language for event-driven applications. Chapter 3 presents the developed points-to analysis for event-driven programs in detail, and chapter 4 evaluates the algorithm as well as its implementation in the nesC compiler. Finally, chapter 5 discusses the discovered limitations of the analysis, outlines ideas for future work, and summarizes the results.

2 State of the Art

This chapter provides a brief overview of the areas of data-flow analysis, points-to analysis, and the nesC programming language, to prepare the points-to analysis algorithm for event-driven programs that is presented in the next chapter. In section 2.1, data-flow frameworks are introduced to statically discover properties that hold at certain program points. Pointer analyses are presented in section 2.2. That section also contains a description of different points-to and alias analyses, before it concludes with the introduction of two important data structures for points-to analyses: location sets and points-to graphs. Finally, the nesC programming language is discussed in section 2.3. NesC is a language to develop asynchronous, event-driven applications and forms the basis of the input language for the points-to analysis algorithm in chapter 3.

2.1 Data-Flow Analysis

Points-to analysis is a data-flow analysis, and that itself is a form of static program analysis. A static analysis infers certain properties of a program without executing it and the result is valid for all possible sets of input data. This is in contrast to dynamic analyses, which execute the program and report only the properties that were observed during the execution, making the result depending on the particular input data. Static program analyses may be as simple as reporting violations of the coding style, e.g., more than one return statement in a function, or as complex as mathematically proving that a program has certain properties. Static analyses in general, and data-flow analyses in particular, are widely used in code-optimizing compilers and program understanding tools.

Data-flow analyses determine how data generated at one point of a program reaches other points of the program and what effect it has there. Typically, these analyses use the technique presented by Kildall [15]: A *control-flow graph* (CFG) is used to represent the program and the data-flow analysis is defined by *data-flow equations* for every node of the CFG. These equations model the effect of the program part represented by the CFG node on the *data-flow values* and are solved by an iterative algorithm. The early and simple analyses were purely *intraprocedural*, that is, they examined each function or procedure of a program in isolation, while today's more complex analyses are *interprocedural*, i.e., they account for the effects of function calls.

A data-flow analysis usually does not analyze a program in form of its source code. A common program representation for intraprocedural analyses is the control-flow graph. A CFG is a directed graph, where the nodes represent statements of the program and the edges model the possible execution orders of these statements. There are two common levels of granularity: A CFG node represents either a single statement or a *basic block*. A basic block is a sequence of statements with the property that, whenever the first statement is executed, the others must be executed as well, in the order they appear in the block and each statement exactly once. The analysis presented in chapter 3 uses a statement-level CFG as program representation, and therefore, from now on, we will only consider such CFGs. Each CFG has two special nodes, an *entry* and an *exit* node. The entry node indicates the beginning of the function, i.e., the point where the control flow enters the function, and the exit node indicates the end of the function, i.e., where the control flow leaves the function. There is an edge from a node n_1 to another node n_2, if and only if the statement represented by n_2 may be executed immediately after the statement represented by n_1. An `if`-statement leads to a branching in the control flow, indicated by two outgoing edges from the node representing the `if`-statement to two distinct nodes. A loop statement, such as `while` or `for`, leads to a *backedge* from the node for the last statement of the loop body back to the node of the loop header. Important concepts with respect to CFGs are *predecessor*, *successor*, and *dominance*. If there is an edge from a node n_1 to a node n_2, n_1 is a predecessor of n_2, and n_2 is a successor of n_1. A node n_1 dominates a node n_2, if every path from the entry node to n_2 passes through n_1. Likewise, a node n_1 postdominates a node n_2, if every path from n_2 to the exit node passes through n_1. Therefore, the entry node dominates all nodes and all nodes are postdominated by the exit node.

Each node n of the control-flow graph, i.e., each statement of the analyzed function, is associated with two *data-flow variables*: In_n and Out_n. The effect of the statement is then expressed by a *data-flow equation* that puts In_n and Out_n in relation. The general form of the equation depends on the *data-flow direction*, and whether the analyzed property must hold either for some path reaching the node or for all paths. In a *forward* analysis, the data flow is in the same direction as the control flow. The values are propagated to a node from its predecessors and merged into the node's In_n variable. Out_n is expressed as a function of the In_n variable. In a *backward* analysis, the data flow is opposed to the control flow. Therefore, the data-flow values are propagated to a node from its successors, are merged into the node's Out_n variable, and In_n is expressed as a function of Out_n. An analysis for a property that must hold on *all paths* reaching the node, merges the incoming values by computing their intersection. Therefore, the resulting value contains only facts that hold along all data-flow paths to the node. An analysis for a property that must only hold on some path reaching the node, also called *any path* analysis, merges the incoming values by computing their union. Thus, the resulting value contains all facts that hold along at least one path to the node. Examples of well-known analyses are [1, 14]:

- *Reaching definitions*: forward, any path

- *Available expressions*: forward, all path
- *Live variables*: backward, any path
- *Anticipable expressions*: backward, all path

Combining the data-flow equations for all statements results in a system of equations that is iteratively solved. All data-flow variables are initialized before the analysis starts. Depending on the direction of the data flow, the initial value at either the entry or the exit node represents the initial knowledge about the analyzed function. This value is called *boundary information*. For example, for the available expressions analysis, which is a forward analysis, the boundary information is at the entry node, and since no expressions were computed at this point, it specifies that no expression is available. After the initialization, the analysis iterates over all statements and applies the flow functions to compute updated data-flow values. This iteration continues until the computed values do not change anymore, i.e., until a fixpoint is reached.

For this iterative algorithm to work, the set of data-flow values is partially ordered by an "is weaker than" relation $\sqsubseteq$. A value x is weaker than y if the assumption of x instead of y is a *safe approximation* with respect to the analysis's goals. We will take *may alias* analysis as an example. If two distinct variable names refer to the same location in memory, these variables are considered aliases. As a may analysis, the analysis must identify all possible alias relations in a function. Let x and y be two variables visible in the analyzed function. Then, $\{alias(x, y)\}$ is a safe approximation for both, $\{alias(x, y)\}$ and $\varnothing$, i.e., in may alias analysis it is always safe to assume an alias relationship. Thus, $\{alias(x, y)\} \sqsubseteq \varnothing$. In *must alias* analysis it is the other way around: it must report definitive aliases only. Therefore, $\varnothing$ is a safe approximation for both, $\{alias(x, y)\}$ and $\varnothing$, i.e., in must alias analysis it is always safe to assume no alias relationship. Hence, $\varnothing \sqsubseteq \{alias(x, y)\}$. To ensure that the iterative algorithm terminates, we require the partially ordered set (poset) of data-flow values to be a *complete lattice*, to satisfy the *descending chain condition*, and the flow functions to be *monotonic*. A monotonic function is order preserving. Therefore, the Knaster-Tarski fixpoint theorem [26] guarantees the existence of a fixpoint and the descending chain condition ensures that a fixed point will eventually be reached.

An interesting point is, when data-flow information is merged. In a *meet over all path* (MOP) assignment, the data-flow values of all paths that reach a node are merged. This leads to the assignment of the strongest safe value to the node's data-flow variable. In contrast to the path-based MOP, the *maximum fixed point* (MFP) assignment is edge-based. It immediately merges the values from all incoming edges. The MFP assignment is weaker than the MOP assignment. Unfortunately, the common algorithms compute the MFP assignment. For the class of *distributive* flow-functions this is not a problem, as in that case, the MFP solution is equal to the MOP solution. For monotonic but non-distributive functions, in general, the

MFP assignment is strictly weaker than the MOP assignment. On the other hand, as shown by Kam and Ullman [12], the problem of finding the MOP solution for general monotonic data-flow frameworks is undecidable.

Two important properties regarding the precision of intraprocedural analyses are *flow sensitivity* and *path sensitivity*. A flow-sensitive analysis considers the control-flow, i.e., the order in which the statements are executed, within the function. It results in individual information for each program point. Flow-insensitive analyses assume that discovered facts hold throughout the entire function. Therefore, each statement must be visited only once. That makes flow-insensitive analysis computationally less complex, but reduces the precision. Path-sensitive analyses take branching conditions into account. In a may points-to analysis, for example, there would be no points-to edge $p \rightarrow$ `NULL` in the data-flow value at the first statement of the "true" branch of `if (p != NULL)`, even if it reached the if-statement. A path-insensitive analysis, in contrast, propagates the same data-flow value to both branches.

As many programs consist of more than one function, an intraprocedural analysis is not enough. *Interprocedural* analyses compute the effects of called functions on the data-flow values at the call site. Therefore, the program representation is extended from CFGs to *supergraphs* or *interprocedural CFGs* [18]. A supergraph consists of the individual CFGs for all functions and represents function calls by *call* and *return* nodes. There is a *call edge* from the call node to the entry node of the called function and a *return edge* from the exit node of the called function to the corresponding return node in the calling function. A supergraph for three functions `main`, `f`, and `g` is depicted in figure 2.1. The individual CFGs were reduced to show just the call and return nodes. The `main` function first calls `f` and then `g`, and `g` also calls `f`. This program representation has the disadvantage that it contains *infeasible* paths, as can be seen in the example. It is possible to enter `f` from `main`, but take the return edge to `g`. The actual execution of the program can not take this path, therefore it is called infeasible.

An analysis that computes distinct information for a function, depending on the calling context, is called *context-sensitive*. In particular, a context-sensitive analysis does not propagate data-flow values along infeasible paths. In contrast, a *context-insensitive* analysis computes the same information for all invocations of a function. Context-sensitive interprocedural analyses were discussed thoroughly by Sharir and Pnueli [24]. They presented two methods to handle function calls. One is a functional approach that computes context-independent summary flow functions for all functions of a program. The summary flow functions are then applied to the data flow information at a call site and yield the data-flow facts that hold after the call in that particular context. The other approach uses the supergraph as presented, but amends the data-flow information with *call strings* to make the interprocedural control flow explicit. Their algorithm then uses the call strings to prevent the propagation of data-flow facts along infeasible paths. Another approach, followed

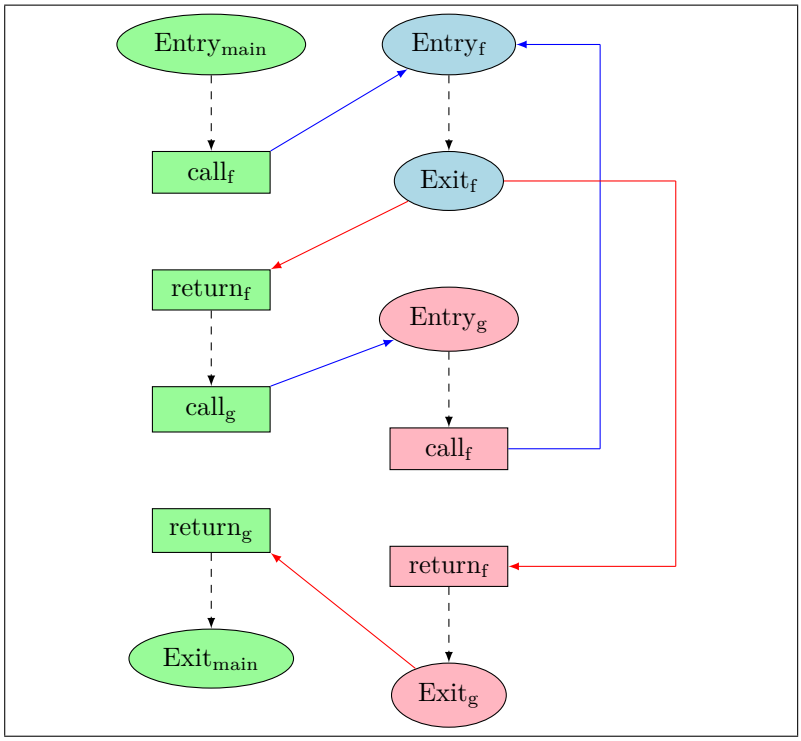

Figure 2.1: A supergraph with three functions: `main`, `f`, and `g`. Call edges are depicted in blue, return edges in red.

by Emami, Ghiya, and Hendren [5], and Whaley and Lam [27] is to use a clone of the called function at every call site, and thereby, to achieve context-sensitive results with an essentially context-insensitive analysis. The analysis presented in this work uses this approach for the interprocedural analysis, as described in section 3.3.

2.2 Points-to Analysis

A points-to analysis is a static program analysis that determines the targets of pointer variables for all possible executions of a given program. The problem of determining pointer targets is, in general, undecidable [4, 17]. Therefore, numerous algorithms have been devised to compute approximate solutions. In his 2001 survey paper [10], Hind counts over seventy-five papers and nine Ph.D. theses on the topic of pointer analysis and these numbers have increased over the last decade. The devised algorithms differ in the precision of the approximation and their computational complexity. A more precise result generally requires a more complex analysis. In addition to the general classification criteria of data-flow analyses, flow sensitivity and context sensitivity, pointer analyses are classified by their modeling of memory. A *field sensitive* analysis distinguishes fields of an aggregate data structure while a

field insensitive analysis treads the entire data structure as one object. Dynamic memory allocations further complicate matters. Simple analyses model the entire *heap* memory as one single object [5]. More advanced analyses differentiate heap blocks by the program point at which they were allocated [28]. The most precise information for heap-allocated data structures is provided by *shape analyses* [8, 23] that create shape graphs to model structures, like trees or lists, on the heap.

```
1: int x, *p, *a;
2: p = &x;
3: a = &p;
```

Figure 2.2: An example program to demonstrate points-to and alias analysis.

A pointer analysis related to points-to analysis is *alias analysis*. While a points-to analysis determines the memory locations a pointer may point to, an alias analysis determines whether two different pointer expressions refer to the same memory location. Hence, a points-to analysis relates pointers to the addresses they point to, whereas an alias analysis relates pointer expressions. We will use the C code in figure 2.2 to demonstrate both analyses. At line 1 one integer variable (`x`), one pointer to an integer variable (`p`), and one pointer to a pointer to an integer variable (`a`) are declared. At line 2 the address of `x` is assigned to `p`. This generates the points-to relation $p \to x$ and the alias relation $\langle *p, x \rangle$. At line 3 the address of `p` is assigned to `a`. This creates the points-to relation $a \to p$ and the aliases $\langle *a, p \rangle$, $\langle * * a, *p \rangle$, and $\langle * * a, x \rangle$. Thus, the result for the points-to analysis after the assignment in line 3 is $a \to p \to x$ and the result for the alias analysis is $\{\langle *a, p \rangle, \langle **a, *p \rangle, \langle **a, x \rangle, \langle *p, x \rangle\}$. As it can be seen in the example, the result of a points-to analysis usually has a more compact representation than the result of an alias analysis. The analysis that will be presented in chapter 3 uses the compact points-to representation, but it is possible to determine the alias relations from this representation.

The two most prominent pointer analyses are the analyses of Andersen [3] and Steensgaard [25]. Both analyses are flow-insensitive, are specified by the means of non-standard type systems, and compute the points-to relations by solving a system of constraints. The analysis of Andersen is a flow-insensitive, context-sensitive, interprocedural may point-to analysis. It computes a mapping from pointer variables to a set of their possible targets. The analysis consists of two phases: constraints for the target sets [2] are generated in the first phase and then the constraints are solved in the second phase. Andersen's analysis considers the direction of assignments, i.e., a statement `p = q`, where `p` and `q` are pointers, leads to a constraint $T_p \subseteq T_q$, where T_p and T_q are the sets of possible targets for `p` and `q` respectively. Such an analysis is called *inclusion-based*. In contrast to Anderson's analysis, Steensgaard's analysis is context-insensitive and less precise. Initially, all pointers are placed in an equivalence class of their own. If the algorithm discovers an assignment between two pointers, it unifies their equivalence classes. Therefore, the direction of assignments

is not considered. In terms of Andersen's analysis, the assignment `p = q` leads to the constraint $T_p = T_q$. Such an analysis is called *unification-based* or *equality-based*. The main argument for Steensgaard's analysis is its low time and space requirement. The required space is linear and the required time almost linear in the size of the input program. More precisely, the time complexity is bounded by $\mathcal{O}(n\alpha(n,n))$, where n is the number of variables in the input program and α is the inverse Ackermann function.

Pointer analyses more related to the analysis presented in this work are flow sensitive. Landi and Ryder [19] presented an alias analysis for programs written in a subset of the C programming language [13]. The analysis is a flow- and context-sensitive, interprocedural analysis that provides a much better precision than previous works. The restrictions on the input programs are that no typecasts are allowed and that all function calls must be explicit, i.e., calls through function pointers are not allowed. Landi and Ryder introduced the concept of *non-visible variables* that are used to represent object names that are not visible in a called function, but may be accessed through pointers. Non-visible variables are similar to *ghost location sets* that will used by our analysis, as described in section 3.3.4. The points-to analysis of Emami, Ghiya, and Hendren [5] lifted these restrictions on the input programs. Rather than computing alias pairs, the analysis computes points-to relations that result in a more compact representation. The analysis uses *abstract stack locations* to model the program's stack memory. Therefore, points-to relations are between stack locations and independent of the types of variables. Hence, typecasts have no effect on the analysis. The algorithm computes an *invocation graph* to model all possible function invocations. This graph incorporates the computed targets for function pointers and is updated when new information is discovered. Using this graph, the analysis handles recursive function calls and calls through function pointers. To achieve context sensitivity, the algorithm maps the available points-to information at a call site into the scope of the called function and analyzes the called function using that information. After the analysis of the called function is completed, the resulting points-to information is mapped back into the name space of the calling function. Emami, Ghiya, and Hendren separate the analysis for data structures on the stack from the analysis of heap-allocated data structures. Their algorithm computes points-to relations only for objects on the stack. All objects on the heap are merged and represented by the symbolic name `heap`. Our analysis adopts the mapping and unmapping of points-to information at call sites and the points-to abstraction from this work. Wilson and Lam [28] presented a points-to analysis that models both heap- and stack-allocated data structures, but does not cover relationships between individual elements of recursive data structures. The algorithm uses *partial transfer functions* (PTF) to describe the behavior of called functions. Therefore, a called function must not be analyzed again for every call site, as long as a suitable PTF was already computed. According to Wilson and Lam this results in a better performance than for the similar analysis of Emami, Ghiya, and Hendren. Our analysis adopts their *location sets*, a representation for

positions within a block of memory.

The analyses presented up until now are for sequential programs only. Our goal is to analyze event-driven programs, and thus, we need a different analysis. Jhala and Majumdar [11] presented a framework for the interprocedural analysis of asynchronous programs. The analysis is a generalization of the IFDS framework of Reps, Horwitz, and Sagiv [21] that computes the precise meet-over-all-valid-paths solution for a problem. Unfortunately, the algorithm is applicable only to problems with a finite set of data-flow facts and distributive flow functions. The flow functions for points-to analyses are non-distributive, and therefore, another approach is required. Rugina and Rinard [22] presented an interprocedural, flow- and context-sensitive points-to analysis algorithm for multithreaded programs. The algorithm computes a conservative approximation of the memory locations a pointer may point to, taking the potential for interference between parallel threads into account. Therefore, it computes a *points-to graph* for every program point, representing the points-to relations at that point. To capture the interference from other threads, the algorithm computes a points-to graph for every thread that specifies the interference information from the parallel threads. In addition to threads, the analysis of Rugina and Rinard handles recursive functions, function pointers, pointer arithmetic, typecasts, and heap- and stack-allocated memory. Furthermore, in the context of threads, it supports *thread private* global variables. Even if this is an analysis for multithreaded programs, it forms the basis of our analysis for asynchronous, event-driven programs. We will model these programs as programs executing multiple *threads of events* in parallel. Therefore, the next chapter provides a detailed description of Rugina and Rinard's algorithm together with the required modifications for event-driven applications written in nesC, which is presented in the next section.

We will conclude the discussion of pointer analyses with a detailed description of two important data structures: location sets and points-to graphs. Location sets were introduced by Wilson and Lam [28] to represent a block of memory and a set of locations in that block. A location set is a triple $\langle n, o, s \rangle$, where n is the name of the block, o is the *offset* within that block, and s is the *stride* that specifies the size of an element in an array. The name is a string of characters, and offset and stride are given in bytes. Thus, a location set represents all locations $\{o + is \mid i \in \mathbb{N}\}$ within block n. A block corresponds to a scalar variable, a structure, an array, or a block of heap memory. The positions of the blocks relative to one another are undefined. Figure 2.3 lists location sets for selected expressions. A scalar variable has an offset and stride of 0, and a field in a structure is represented by its offset in bytes from the beginning of the structure. An entire array is represented like a scalar variable, but an element of the array has a stride equal to the element size. Location sets for array elements have no offset, and therefore, all elements are represented by the same location set. Thus, the analysis does not differentiate array elements, while it differentiates fields of structures. The location set that represents a field in a structure that is an element of an array has the stride equal to the size of the structure and an offset equal to the distance from the begin of the structure to the begin of the field. Hence, as

scalar array elements, the structures are not differentiated, but the individual fields of the structures still are. A special case are structs that contain arrays. As C and nesC do not provide bounds checking, the analysis conservatively assumes possible out-of-bounds accesses to arrays, and therefore models arrays in structures as if they would overlap with the entire structure. Hence, the offset is less than the stride and that is enforced by computing the offset modulo the stride. For expressions where the position within the block of memory is unknown, the stride of the location set is set to one.

Expression	Location Set
`var`	$\langle var, 0, 0\rangle$
`struct.f`	$\langle s, o, 0\rangle$
`arr`	$\langle a, 0, 0\rangle$
`arr[i]`	$\langle a, 0, s\rangle$
`arr[i].f`	$\langle a, o, s\rangle$
`struct.f[i]`	$\langle a, f\%s, s\rangle$
`*(p + Z)`	$\langle p, 0, 1\rangle$

Figure 2.3: Location sets for some expressions: `var` is a scalar variable, `struct` is a structure with a field `f`, `arr` is an array, `p` is a pointer, and `Z` is an integer; o is the offset of `f` from the beginning of the structure and s is the size of an array element.

Points-to graphs represent the relations between pointers and their targets. All variables and functions are represented by location sets and the set of all location sets is the set of nodes in a points-to graph. If a pointer may point to an object, the points-to graph contains a directed edge from the location set corresponding to the pointer to the location set corresponding to the object. Furthermore, the points-to graph contains a special location set called `unk` that represents all unknown memory locations. For every possibly uninitialized pointer, there is an edge from the corresponding location set to `unk`. An individual edge in the points-to graph represents a may point-to relation, but the entire graph provides both may and must point-to information. If there is a single outgoing edge from a node p to a node x then `p` must definitively point to `x`. On the other hand, if there are two or more outgoing edges from a node p then `p` may point to either location. If there is an edge to `unk`, the pointer may be uninitialized and if it is the only edge, the pointer must be uninitialized. Figure 2.4 depicts a points-to graph. The two outgoing edges from a indicate that `a` may either point to `p` or to `q`. The single outgoing edge from p indicates that `p` must point to `x`. There are edges from q to y and to `unk`. Therefore, `q` may either point to `y` or `q` may be uninitialized, and thus point to some unknown location.

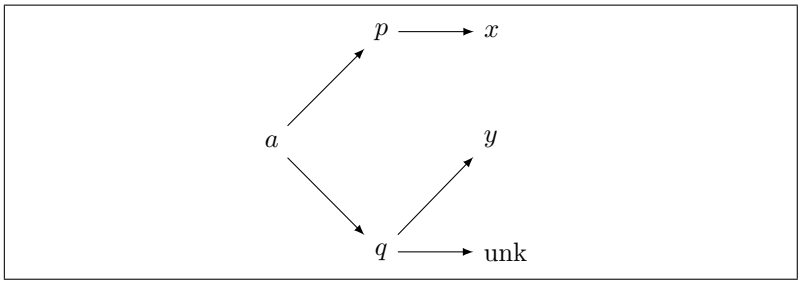

Figure 2.4: A points-to graph for the pointers `a`, `p`, and `q`, the variables `x` and `y`, and the location set indicating unknown locations.

2.3 NesC

NesC [6] is a programming language for *component-based*, *event-driven* applications. It is an extension and restriction of the C programming language [13]. The main additions are *commands*, *events*, and *tasks*. NesC was designed specifically to reimplement [7] the TinyOS operating system [9] for sensors in distributed, wireless sensor networks. The characteristic of such wireless sensors is that they are small, low-power devices that are driven by their interaction with the environment. Therefore, nesC's execution model is event-driven. An application waits for external events, such as timers, received radio messages, or sensor readings. When an event occurs, the corresponding *event handler* is invoked to process the event. Once processing is complete, the application waits for the next event to occur. Triggered by changes in the environment, a new event may occur while another is still processed. Being designed for low power consumption, the sensors are uniprocessor systems, and therefore, can not execute program code in parallel. However, through the interleaved execution of event handlers, it is possible to simulate parallel execution.

A nesC application consists of several components. Components are used to structure the application, to group related functionality, and to encapsulate the state of different parts of the system. A component offers its services through one or more interfaces that it implements. Interfaces in nesC are bidirectional; additionally to the provided services they specify functionality that the interface user must implement. The interface user has to implement callback functions that are used by the interface provider to signal the completion of the requested service. Commands are services provided by the interface provider and are declared with the `command` keyword. Events are callback functions that the interface user must provide, and are declared with the `event` keyword. Operations that do not complete immediately are *split-phase*. The interface user starts an operation by calling the corresponding command. The interface provider schedules the operation for future execution and returns the control immediately back to the user. After the operation was executed, the interface provider signals the completion and returns the result to the user through an event. This resembles the handling of hardware events and adopts

the event-driven paradigm as fundamental concept of nesC.

```
module ExampleC {
  uses interface Boot;
  uses interface Timer<TMilli>;
  uses interface Read<uint16_t>;
} implementation {

  uint16_t lastValue = 0;

  event void Boot.booted() {
    call Timer.startPeriodic(1000);
  }

  event void Timer.fired() {
    call Read.read();
  }

  event void Read.readDone(error_t result, uint16_t val) {
    if (result == SUCCESS) {
      lastValue = val;
    }
  }
}
```

Figure 2.5: An example of split-phase operations.

We will use the nesC module in Figure 2.5 to demonstrate nesC's concepts of components (encapsulation of state), interfaces, and the event-driven execution. The module is named `ExampleC` and uses three interfaces: `Boot`, `Timer`, and `Read`. `Timer` and `Read` are parametrized interfaces. The parameter `TMilli` specifies that the used timer should have millisecond resolution and the parameter `uint16_t` specifies that the sensor accessed through the `Read` interface provides its values as 16 bit unsigned integers. The module stores the last read value in a variable that is only visible within the module. If the value should be accessed from another module, the `ExampleC` module must provide access to it through its interface. This demonstrates how modules provide encapsulation of internal state. The module implements three event handlers: `Boot.booted` from the `Boot` interface, `Timer.fired` from the `Timer` interface, and `Read.readDone` from the `Read` interface. TinyOS signals the `Boot.booted` event after the sensor node was started and the basic subsystems have been initialized, so that the higher-level components can perform their initialization. The example module uses this event handler to call the `Timer.startPeriodic` command from the `Timer` interface to start a timer that signals the `Timer.fired` event every second. After this initialization, the module waits for the timer event. When the specified time of one second expired, the timer signals the `Timer.fired` event, and thereby transfers the control to the event handler in the module. The handler

for `Timer.fired` calls the `Read.read` command from the `Read` interface to acquire a value from the sensor. The command can not immediately return the requested value. Therefore, it initiates whatever is necessary to get the value from the sensor and then returns the control back to the `Timer.fired` event handler. The `Timer.fired` event handler has nothing else to do, so it returns the control to its caller, somewhere in the module implementing the `Timer` interface. Once the value is available from the sensor, the sensor module signals it to the `ExampleC` module by invoking the `Read.readDone` event handler. The event handler checks whether the read operation was successful and if it was, stores the value in the `lastValue` variable. After that, the module waits for the next timer event, to trigger the next read of the sensor's value.

NesC divides applications into *synchronous* and *asynchronous* code. Asynchronous code is code that is reachable from an *interrupt handler*. An interrupt handler is invoked immediately when the interrupt occurs. Thus, it may execute between any two statements in the application. Synchronous code is not reachable from interrupt handlers, and hence, is executed only when it is invoked from other synchronous code. Asynchronous code may lead to *race conditions*, where some shared state is unexpectedly modified between two accesses from other code. In nesC, the transition from asynchronous execution to synchronous execution is possible through *tasks*. Tasks allow the deferred, synchronous execution of code. A task is declared with the `task` keyword and scheduled for future execution with the `post` statement. Tasks are atomic with respect to other synchronous code. Therefore, once a task started to execute, it will run to completion.

The points-to analysis presented in chapter 3 makes four assumptions about the analyzed programs. Since the analysis processes the top-level module only, those assumptions must hold for that module, but not necessarily for other modules. The first assumption concerns dynamic memory. The NesC manual [6] states that nesC does not support dynamic memory allocation, but such allocations are still possible. Following the nesC manual and programming hint 3 in TinyOS programming [20]:"Never use malloc and free", the analysis requires that there are no dynamic memory allocations in the analyzed program. Thus, the analysis does not model the heap memory. Another restriction follows from programming hint 2: "Never write recursive functions." Assuming non-recursive functions limits the application's stack usage and simplifies the interprocedural points-to analysis. These two assumptions are best practices in nesC. The following two assumptions are restrictions that go beyond that. For one, there must be no interference from asynchronous events. All events and tasks are assumed to be atomic with respect to all other code. Second, a called command must not have any effect on the point-to relations in the analyzed module. As only one module is analyzed, these modifications would not be detected and the analysis would be unsound.

3 The Analysis: Algorithm and Implementation

Pointer analysis and event-driven programs have been introduced in the last chapter. This chapter introduces the points-to analysis algorithm for event-driven programs. First, section 3.1 gives an overview of the algorithm, then sections 3.2 to 3.4 introduce the different parts of the analysis, namely intraprocedural, interprocedural, and event-sensitive analysis. These three sections all have the same layout, starting with the considered input language, followed by an example of what to expect from the analysis. Then the intermediate representation—the representation of the program that the analysis operates on—is presented, before the data-flow equations are given. The properties of the presented algorithm are summarized at the end of each of these sections. Finally, section 3.5 discusses the integration of the analysis into the nesC tool chain.

3.1 Overview

The pointer-analysis algorithm for event-driven programs is built upon Rugina and Rinard's pointer-analysis algorithm for multithreaded programs [22]. The algorithm of Rugina and Rinard computes a safe approximation of the points-to graph for every program point, including any points-to edges generated due to the interleaved execution of other threads. Their level of granularity is a single statement, i.e., the execution of one thread may be interrupted after any statement, then one or more statements from other threads may be executed, before the next statement of the first thread is executed. Our analysis addresses nesC programs, where the execution model is based on run-to-completion tasks and events. Since we ignore hardware-event handlers and asynchronous events in general and additionally assume that the analyzed pointers are modified by synchronous code only, our granularity, with respect to concurrent execution, is a single event. Hence, at the top level, we will model an event as a single statement in the analysis for multithreaded programs. An event may trigger the future execution of another event, which in turn may trigger the future execution of yet another event, and so on. These event sequences are modeled as individual threads. Whenever an event directly triggers the future execution of more than one event, the future events are modeled as concurrently executing threads. The presented algorithm performs an interprocedural analysis of

an application's top-level module and, as Rugina and Rinard's algorithm, is flow- and context-sensitive.

The following sections give a detailed presentation of the analysis algorithm and the developed model of event-driven programs as multithreaded programs. Section 3.2 presents the intraprocedural analysis that computes a points-to graph for every program point within a single function or event without considering the effect of function calls. In section 3.3, the intraprocedural analysis is extended to support function calls in the input language and to consider the effects of the called function on the points-to graphs. Section 3.4 completes the analysis algorithm by adding events to the input language and considering the effect of their non-deterministic execution order.

3.2 Intraprocedural Analysis

This section presents the intraprocedural points-to analysis, the basis of the overall algorithm. It analyzes the points-to relations within a single function or event. The difference between functions and events is in the time their execution starts; once they have started, their behavior is the same. Therefore, in this section, we use the term function to refer to both functions and events. NesC's execution model does not allow any concurrent code execution within a function or an event. Furthermore, once the execution of a function started it is guaranteed to finish its execution without interference by any other function. Therefore, the simplest form of intraprocedural analysis can omit support for parallel execution constructs. As we will see in section 3.4, once we introduce parallelism at the event level, we have to modify the data-flow equations for intraprocedural analysis to generate and propagate additional information. Nevertheless, for clarity, we leave this additional information out in this section and introduce it once it is needed.

3.2.1 Language

The supported input language is a subset of both the C and nesC language. It can be divided into two parts, assignment statements and control-flow statements.

Address-of Assignment	`x = &y`
Constant Assignment	`x =` *const*
Copy Assignment	`x = y`
Load Assignment	`x = *y`
Store Assignment	`*x = y`

Figure 3.1: Basic assignment statements.

The analysis supports five basic assignment statements, which are given in figure 3.1. The effect of these statements on the points-to graph is shown in figure 3.2. This will be explained in detail with the help of the example in the next section. These basic statements permit the address-of and dereference operator to appear in assignments, as it is allowed by the C language specification, with the constraint that only one such operator can occur in the statement. This is a constraint on the input language, but it is not a limitation in expressiveness. Every more complex assignment statement can be reduced to these basic forms, using temporary variables.

Before	Assignment	After
$x \quad y$	`x = &y`	$x \rightarrow y$
x	`x =` *const*	$x \rightarrow ?$
x $y \rightarrow a$	`x = y`	x $\searrow$ $y \rightarrow a$
x $y \rightarrow z \rightarrow a$	`x = *y`	x $\searrow$ $y \rightarrow z \rightarrow a$
$x \rightarrow z$ $y \rightarrow a$	`*x = y`	$x \rightarrow z$ $\downarrow$ $y \rightarrow a$

Figure 3.2: Effects of the basic assignment statements on the points-to graph.

Besides the basic assignment statements, the input language supports branches and loops in the control flow. Branches can be expressed with `if` statements and loops with `for` statements. This is, again, only a limitation in the input language; `if-else` and `while` statements can be transformed into `if` and `for` statements. Furthermore, as will be described in section 3.2.3, the algorithm works on the CFG as an intermediate representation. Therefore, once the CFG is available, the data-flow equations are independent of the statements used to describe the control flow.

3.2.2 Example

After the last section introduced the input language, this section gives two example programs to show what to expect from the analysis algorithm that is presented in the next section. One example is to demonstrate the effect of the basic assignment statements and the other to show how control-flow branches and merges influence the points-to information.

The simple nesC module in figure 3.3 is our first example. It declares two 8 bit wide unsigned integer variables (`x`, `y`), three pointers to such variables (`p`, `q`, `r`), and one pointer to a pointer to such a variable (`a`). The event `Boot.booted` is the entry point to the application. The CFG for this function is depicted in figure 3.4 together with the known points-to graph before and after every statement. The statements

```
module ExampleC {
  uses interface Boot;
} implementation {
  uint8_t x, y;
  uint8_t *p, *q, *r;
  uint8_t **a;

  event void Boot.booted() {
1:  x = 0;
2:  y = 0;

3:  p = &x;
4:  q = &y;
5:  a = &p;

6:  r = q;

7:  q = *a;

8:  *a = r;
  }
}
```

Figure 3.3: A program to demonstrate the basic assignment statements.

at lines 1 and 2 initialize the variables `x` and `y` to `0`. This generates no edges in the points-to graph, as `x` and `y` are no pointers. The statements at lines 3, 4, and 5 are address-of assignments. At first, `p` is assigned the address of `x`, and thus, `p` points to `x`. This adds the edge $p \to x$ to the points-to graph. Then the address of `y` is assigned to `q`, generating the edge $q \to y$. Finally, the address of the pointer `p` is assigned to the pointer-to-a-pointer `a`. Therefore, we get an edge $a \to p$ and thus the subgraph $a \to p \to x$. From this subgraph and the transitivity of the points-to relation we can also get the information that `**a` is an alias of `x`. The statement at line 6 is a copy assignment. `r` is assigned the address stored in `q`. As `q` holds the address of `y` the points-to edge $r \to y$ is added to the graph and now both `q` and `r` point to `y`. At line 7 is a load assignment. The pointer `q` is assigned the value of the target of `a`; `a` points to `p` and `p` points to `x`, therefore, `q` now points to `x`. This removes the edge $q \to y$ and adds the edge $q \to x$ instead. Finally, at line 8 is a store assignment. The target of `a` is assigned the value of `r`. `a` points to `p` and `r` points to `y`. Thus, `p` now points to `y`. Before this assignment `p` pointed to `x`, so the edge $p \to x$ is removed and the edge $p \to y$ is added.

The other example is given in figure 3.5. It demonstrates the effects of branching and merging in the control flow. The general conditions are the same as in the example before. The annotated CFG is in figure 3.6. As the effects of the basic as-

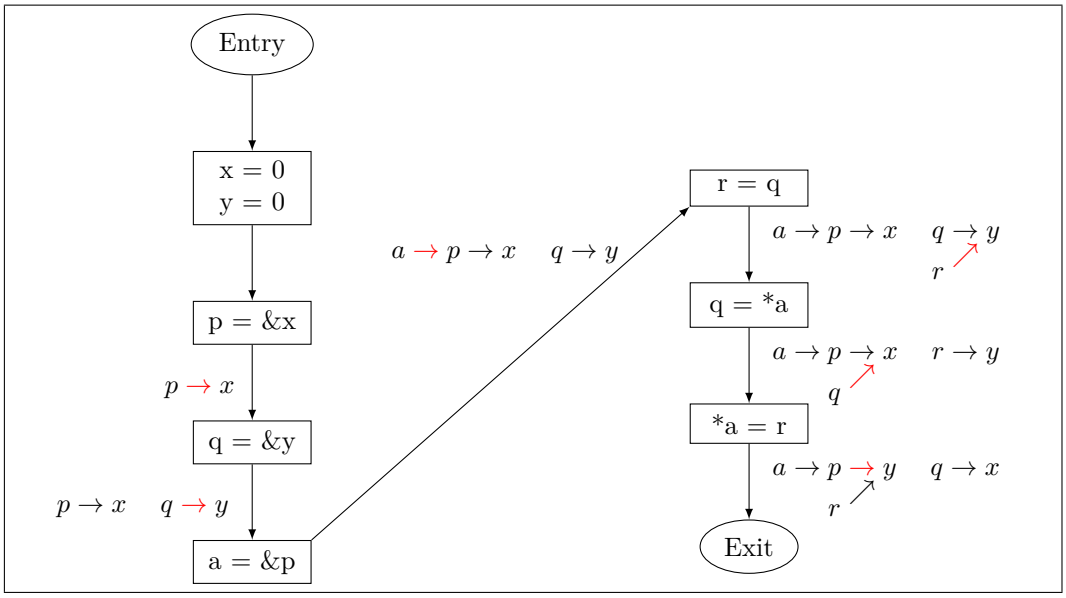

Figure 3.4: Pointer-analysis result for the example program with basic assignment statements in figure 3.3.

signment statements were already shown in the last example, sequential assignments are combined in a single CFG node for space reasons. Up to and including line 6 this example is equal to the example before. Therefore, we start the description with the `if` statement at line 7. The `if` statement leads to a branch in the control flow, so the statement that is executed afterwards is either the assignment in line 7 or the one in line 8. In the control-flow graph, this is depicted by the two outgoing edges from the corresponding node. The analysis does not consider the branching condition. Therefore, it does not know whether it is impossible to take a specific branch or what additional constraints the `if` statement imposes on the points-to graph for a given branch. Thus, the points-to graph that arrives at the `if` node is propagated to both branches unchanged. If the right branch is taken, the pointer-to-a-pointer `a` is assigned the address of `q`. This removes the points-to edge $a \to p$ and adds the edge $a \to q$. The interesting point is when the two branches are merged together — in the CFG depicted by the circle-shaped node that does not correspond to any statement. The analysis must unify the two different points-to graphs such that the single resulting points-to graph is a safe approximation for both input graphs. The points-to edges $p \to x$ and $q \to y$ are in both input graphs, so they definitively must also be in the unified graph. The points-to graph from the left branch also contains the edge $a \to p$, but this edge is missing in the graph from the right branch. The graph from the right branch in turn contains an edge $a \to q$ that is missing in the graph from the left branch. As the points-to graph must include an edge for every *possible* target of a pointer at a given program point, both edges must be included. Thus, the merged set of points-to edges in the graph is the union of the two input

```
module ExampleC {
  uses interface Boot;
} implementation {
  uint8_t x, y;
  uint8_t *p, *q;
  unit8_t **a;

  event void Boot.booted() {
1:  x = 0;
2:  y = 0;

3:  p = &x;
4:  q = &y;
5:  a = &p;

6:  if (cond)
7:    a = &q;

8:  *a = p;
  }
}
```

Figure 3.5: A program to demonstrate branching and merging.

edge sets. Up to now, the points-to graph contained at most one outgoing edge from every pointer, allowing to uniquely answer the question where a pointer *must* be pointing to. After the merge, the points-to graph contains two outgoing edges from `a`. Therefore, the question where `a` points to can no longer be answered definitively. It is only possible to give the weaker answer that `a` *may* point to either `p` or `q`. So we have a combined may- and must-point-to analysis. Furthermore, this weaker answer has consequences for the store assignment in line 8. As the exact target of `a` is unknown, all possible targets must be updated. This leads to the edges $p \to x$ and $q \to x$. The edge $p \to x$ is already in the graph, therefore, only $q \to x$ is added as a new edge. Moreover, `a` may not point to `q`, so the edge $q \to y$ must be kept in the points-to graph.

3.2.3 Intermediate Representation

As already noted in section 3.2.1, the analysis does not analyze the program as it was written in the input language, but rather uses a control-flow graph as intermediate representation. Therefore, before the analysis starts, such a CFG must be created for every function from the program's source code. The begin of the function is indicated by an entry node, called "Entry", that has no incoming edges, but an outgoing edge, leading to the first statement of the function. Likewise, the end of

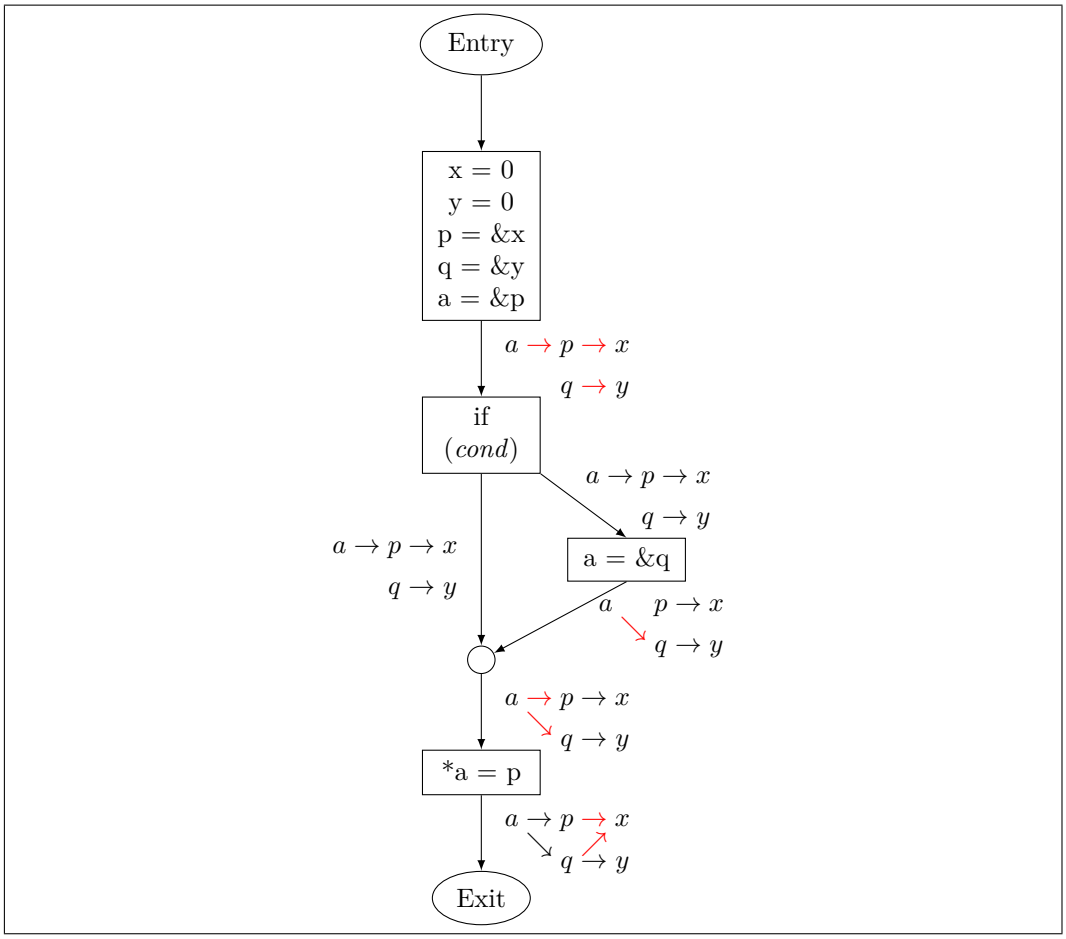

Figure 3.6: Pointer-analysis result for the example program with branching and merging.

a function is indicated by an exit node, called "Exit", that has an incoming edge from the last statement of the function, but no outgoing edges. The used control-flow graph is a *statement level* CFG. That means that every CFG node represents exactly one statement of the program. More precisely, every node of the CFG, except the entry and exit nodes, contains a single basic assignment statement, as listed in figure 3.1, or a control flow statement. Although nodes for control-flow statements are present in the CFG, the control-flow statements are ignored by the analysis algorithm. The control flow is determined solely by the edges of the CFG, and thus, the analysis is path insensitive.

The analysis starts at the entry node with the *boundary information* that all pointers are uninitialized. The points-to information is propagated down the edges of the control-flow graph and is updated at every node according to the data-flow equations that will be given in the next section. Once a fixed point is reached, i.e., the repeated

application of the data-flow equations does not change the points-to graphs anymore, the algorithm terminates.

3.2.4 Equations

This section presents the main part of the intraprocedural analysis: the data-flow equations used to update the points-to graph at an assignment statement. As the intraprocedural analysis, at this time, is not required to handle the concurrent execution of threads, only a basic form of the equations is given here. Once parallelism is introduced at the event level in section 3.4, we will update the equations for intraprocedural analysis in section 3.4.4. This section starts with a definition of points-to information and shows how this forms a lattice, before defining the data-flow equations and listing the *gen* and *kill* sets for the basic assignment statements.

In this first stage, the points-to information at any program point consists only of the points-to graph at that point. This leads to the following straightforward definition:

Definition 3.1. Let L be the set of all location sets in the program and $P = 2^{L \times L}$ the set of all points-to graphs for these location sets. The *points-to information* $\mathrm{PI}(p)$ at a point p in the program is the points-to graph $C \in P$ at that point.

The current points-to graph is represented by the set of its edges: $C \subseteq L \times L$. We order points-to graphs by *set inclusion*:

$$C_1 \sqsubseteq C_2 \iff C_1 \subseteq C_2$$

This is a partial ordering, and thus, $(P, \sqsubseteq)$ is a poset. Furthermore, $(P, \sqsubseteq)$ forms a lattice with *set union* as meet operator:

$$C_1 \sqcup C_2 := C_1 \cup C_2$$

Let *Stat* be the set of all statements in the analyzed function and $[\![\ ]\!] : \mathit{Stat} \to (P \to P)$ a functional that assigns a transfer function $f \in P \to P$ to every statement $st \in \mathit{Stat}$. For basic assignment statements, the functional $[\![\ ]\!]$ is shown in figure 3.7 and defined in terms of *gen* sets, *kill* sets, and a *strong* flag as given in figure 3.8.

$$[\![st]\!]C = C', \text{ where:}$$

$$C' = \begin{cases} (C - \mathit{kill}) \cup \mathit{gen} & \text{if } \mathit{strong} \\ C \cup \mathit{gen} & \text{if not } \mathit{strong} \end{cases}$$

Figure 3.7: Data-flow equations for the intraprocedural analysis.

Address-of Assignment	$x = \&y$	$kill := \{x\} \times \text{deref}(\{x\}, C)$ $gen := \{x\} \times \{y\}$ $strong := x.stride = 0$
Constant Assignment	$x = const$	$kill := \{x\} \times \text{deref}(\{x\}, C)$ $gen := \{x\} \times \{\texttt{unk}\}$ $strong := x.stride = 0$
Copy Assignment	$x = y$	$kill := \{x\} \times \text{deref}(\{x\}, C)$ $gen := \{x\} \times \text{deref}(\{y\}, C)$ $strong := x.stride = 0$
Load Assignment	$x = *y$	$kill := \{x\} \times \text{deref}(\{x\}, C)$ $gen := \{x\} \times \text{deref}(\text{deref}(\{y\}, C), C)$ $strong := x.stride = 0$
Store Assignment	$*x = y$	$kill := \text{deref}(\{x\}, C) \times \text{deref}(\text{deref}(\{x\}, C), C)$ $gen := \text{deref}(\{x\}, C) \times \text{deref}(\{y\}, C) - \{\texttt{unk}\} \times L$ $strong := \text{deref}(\{x\}, C) = \{z\}$ and $z.stride = 0$

Figure 3.8: Basic pointer assignment statements and the corresponding data-flow sets.

The *gen* set contains the points-to edges that are generated by the statement and the *kill* set contains the edges that are removed, i.e., replaced by the generated ones. The *strong* flag plays an important role in the points-to graph's update. If it is true, the pointer that is assigned a new value can be precisely determined. As the assignment overrides the former value of the pointer, the corresponding edge can safely be removed from the points-to graph. This is called a *strong update* and reflects exactly what the program is doing. The *strong* flag is false, if the pointer can not be determined exactly, i.e., if there is more than one pointer that may be updated by the statement. As it is unclear which pointer will be updated, it is consequently also unclear which pointers keep their original value. Therefore, the analysis must assume that all pointers may be pointing to the new target, while at the same time they may still point to the old one. Thus, the newly generated edges from the *gen* set are merged into the points-to graph, but the edges from the *kill* set are not removed. This is called a *weak update* and is only an approximation of the actual assignment in the program. After both forms of update, the set of actual points-to relations, in any program execution, is a subset of the points-to relations expressed by the computed points-to graph. Hence, the approximation is safe.

The definition of the *gen* and *kill* sets uses the *deref* function

$$\text{deref} : 2^L \times P \to 2^L$$
$$\text{deref}(S, C) = \{y \in L \mid \exists x \in S \,.\, (x, y) \in C\}$$

to model pointer dereferences. Given a set of location sets, S, that contains the location sets corresponding to the dereferenced pointers and the current points-to graph C, $deref(S, C)$ yields a set of all location sets that the pointers from S

may point to. Since the dereference of a pointer at an unknown location yields an unknown location, *deref*($\{\texttt{unk}\}, C$) = $\{\texttt{unk}\}$, for all points-to graphs C.

The address-of assignment is straightforward. After the assignment `x = &y`, `x` points to `y`, so the *gen* set contains exactly the edge $x \rightarrow y$. The definition of the *kill* set and the *strong* flag is the same for the first four assignment forms in figure 3.8. As the pointer `x` is updated, it no longer points to its former target. Therefore, the *kill* set contains all edges of the points-to graph that start at `x`. Whenever the stride of the location set is 0, i.e., the pointer is not an element of an array, the location set unambiguously identifies a single pointer, and thus, a strong update is performed. Otherwise, the location set represents all array elements, but only one element is assigned a new value. Therefore, the resulting update must be weak.

Constant assignments are assignments of a constant address to the pointer. The most common example is the assignment `x = NULL`, to indicate that the pointer does not point to a valid location. In general it is impossible to say where a given address refers to. Hence, the analysis generates a points-to edge $x \rightarrow \texttt{unk}$, indicating that the target is unknown.

A copy assignment `x = y` takes the address of the target of `y` and assigns it to `x`. Thus, the analysis has to take all location sets for possible targets of `y` and generates a points-to edge from `x`'s location set to the target location set.

Load assignments are a bit more complicated. For the assignment statement `x = *y`, as for the assignments before, the location set for `x` is unambiguously determined. However, the dereferencing on the right-hand side of the statement requires a second application of the *deref* function. The first application determines the location sets for the targets of `y`, but after the assignment, `x` points to the target of the target of `y`. Therefore, a second application of *deref* is necessary, with the set of location sets resulting from the first application. This second application yields all possible new targets for `x`, and thus, the new points-to edges can be generated.

Store assignments are even more complicated, and different from the other assignment statements. In a store assignment `*x = y`, the location set for the left-hand side is not directly available and there may be even more than one. Thus, the location sets for the left-hand side must be determined by applying the *deref* function to the location set for `x`. Each of these location sets forms the source of points-to edges, in both *gen* and *kill* sets. The new target of `*x` is the target of `y`, so the targets of the points-to edges in the *gen* set are determined by applying *deref* to `y`. Since *deref*($\{x\}, C$) may yield `unk` and we do not want to add irrelevant edges *starting at* `unk` to the points-to graph, all edges starting at `unk` are not part of the *gen* set. Nevertheless, an assignment to an unknown location is dangerous, as it may update any pointer in the program. Therefore, the analysis should conservatively add points-to edges from every location set to all possible targets to the points-to graph. However, this would result in extremely imprecise results, even for programs that ensure that the dereferenced pointer points to a valid location before updating

that location. This results from the path-insensitivity of the analysis. The common `if (x != NULL)` tests are not considered by the analysis, and thus, the analysis optimistically assumes that there are no updates through pointers to unknown locations to yield a more precise result. The *kill* set consists of all edges starting at the possible location sets for `*x`, as these may potentially be overridden. Whether the edges from the *kill* set are actually removed from the points-to graph is, again, determined by the *strong* flag. The computation of the *strong* flag for store assignments, correspondingly, depends on the stride value of the left-hand side's location set. The difference here is that there may be more than one location set for the left-hand side. In case there is, the analysis can not definitively determine which one is updated, so that the corresponding edges could be removed. Hence, if *deref*$(\{x\}, C)$ yields a set with cardinality greater than one, the update must be weak.

The equations in figure 3.7 and figure 3.8 resemble the ones given by Rugina and Rinard with two differences. The nesC language does not support threads or has any construct to express the parallel execution of code. Therefore, Rugina and Rinard's `par`-construct is not required for the intraprocedural analysis, and hence, the I and E sets from their definition of multithreaded points-to information can be omitted. Furthermore, while not preventing dynamic memory allocations, e.g., through `malloc()` and `free()`, nesC itself does not support these. In fact, the usage of dynamically-allocated memory in nesC programs is strongly advised against. For this reason, the analysis does not support pointers to memory locations dynamically allocated on the heap. This decision results in a simplified definition of the *strong* flag, as the heap-related part is left out.

3.2.5 Properties

This section discusses some properties of the algorithm and concludes the presentation of the intraprocedural part of the analysis. Since the intraprocedural analysis resembles the one of Rugina and Rinard less support for some language constructs, some properties can be transferred directly. The analysis propagates the points-to information along the edges of the control-flow graph and maintains a separate points-to graph for every program point in the analyzed function. Thus, the presented analysis is flow sensitive. Branches in the control flow are handled without considering the branching condition. The points-to information that reaches the CFG node for a control-flow statement is propagated unchanged into both branches. Therefore, the analysis is path insensitive. The data-flow equations are monotonic, but, as is typical for pointer-analyses, non-distributive.

A pointer `p` points to object `x` if `p` contains the address of `x`. The analysis represents this relationship by a points-to edge $p \to x$. Thus, to be sound, the analysis must yield a points-to graph at program point n that contains a points-to edge $p \to x$ whenever it is possible that `p` contains the address of `x` at n. More formal: `p == &x` at program point $n \Rightarrow \{p \to x\} \subseteq \mathrm{PI}(n)$. For `p` to contain the address of `x`, it must

have been assigned to `p` at some point in the program. This assignment is either directly by an address-of assignment or the address is copied from another pointer by a copy, load, or store assignment. The data-flow equations for these statements generate the corresponding points-to edges in the points-to graph, as long as an address is not written to the target of an uninitialized pointer. Therefore, whenever a points-to relation is created in the program, a corresponding edge is generated in the points-to graph. What remains to be shown is that these edges are also present in the points-to graphs at all program points where the pointer may still contain the address of the object, i.e., no other address was assigned to the pointer in between. The data-flow information is propagated along the edges of the control-flow graph. Hence, all points-to graphs at program points reachable from the point of the assignment contain the corresponding points-to edge, unless it was removed from the graph. At a branch in the control flow, the information that reaches the branching point is propagated into all branches unchanged and when the control flow is merged from different branches, the points-to graph after the merge contains all edges that are present in any graph reaching the merging point. Thus, the only way to remove an edge from the points-to graph is that it is contained in the *kill* set of a strong update. Since an update is strong only if the updated pointer can be identified unambiguously, a points-to edge is removed from the points-to graph only if some other address was definitively assigned to the pointer. Furthermore, let us consider the points-to graph at a specific program point. The points-to graph at the entry point represents the boundary information and does not change during the analysis. For all other program points, the points-to graph initially contains no edges. Every time the algorithm analyzes the statement at that program point, it may only add further edges to the points-to graph. Therefore, the descending chain condition holds, and as the analysis generates and propagates the proper points-to edges and removes them only by a strong update, the analysis is sound.

To derive an upper bound on the complexity of the algorithm we will consider the CFG for the analyzed function. As the analyzed function consists of a finite number of statements, the number of nodes in the CFG is also finite. Let n be the number of nodes in the CFG. All location sets appear in a statement of the program. Therefore, there are at most $\mathcal{O}(n)$ distinct location sets, and thus the points-to graphs have at most $\mathcal{O}(n^2)$ edges. The points-to information could be distinct for every program point, i.e., there is one points-to graph maintained at every CFG node. Hence, all points-to graphs combined have at most $\mathcal{O}(n^3)$ edges. The points-to graph at a program point, except the entry point, initially contains no edges. As the meet operation is set union, all subsequent analysis steps may only add new edges to a points-to graph. The algorithm terminates after processing at most n CFG nodes without adding an edge to a points-to graph. This results in an upper bound of $\mathcal{O}(n^4)$ analysis steps that must be carried out. Therefore, the asymptotic complexity is equal to the one of the algorithm for multithreaded programs. Finally, the algorithm terminates. This follows with arguments similar to the ones from the complexity considerations as well as from the termination property of Rugina and

Rinard's algorithm.

3.3 Interprocedural Analysis

The intraprocedural part of the analysis was defined in the last section. This section extends the analysis to support function calls. Functions are part of many high-level programming languages and many programs make use of them. In particular, functions are a part of the nesC programming language, and all non-trivial TinyOS applications contain function calls, either directly or as command calls. To analyze a function that calls another function, the algorithm must determine the effect of the invoked function on the invoking function, or more precisely, the effect on the points-to graph at the *call site*.

3.3.1 Language

The input language is an extension of the language specified in section 3.2.1. In addition to basic assignment statements and control-flow statements, the input language now supports statements to declare and invoke functions. The syntax follows the syntax of the C programming language and of nesC. A function is called by its name and a list of parameters in parentheses. The invocation of a function `f` without parameters is therefore written as `f()`.

Following programming hint 2 in TinyOS Programming [20], to never write recursive functions, the analyzed programs are not allowed to have functions that invoke themselves. Furthermore, a function `f` is not allowed to call another function that directly or indirectly leads to a second invocation of `f`.

3.3.2 Example

After extending the input language, we will, again, demonstrate, what to expect from the analysis, before presenting the extended algorithm in the next sections. The source code for the example is given in figure 3.9. The program is a nesC module that consists of three 8 bit unsigned integer variables (`x`, `y`, `z`), two pointers to such variables (`p`, `q`), and three functions (`Boot.booted`, `f`, `g`). `Boot.booted` is the entry point of the application. The corresponding supergraph is pictured in figure 3.10. The solid black edges depict the intraprocedural control flow, while the dashed blue edges illustrate the interprocedural control flow. Note that, other than usual, the call nodes are not split into call and return nodes. This is to resemble the intermediate representation that will be described in the next section. Furthermore, there are two instances of the CFG corresponding to the function `f`, named f and f'.

```
module ExampleC {
  uses interface Boot;
} implementation {
  uint8_t x, y, z;
  uint8_t *p, *q;

  void f();
  void g();

  event void Boot.booted() {
    p = &x;

    f();
    g();
  }

  void f() {
    p = &y;
  }

  void g() {
    p = &z;
    q = &z;
    f();
  }
}
```

Figure 3.9: A program to demonstrate the interprocedural analysis.

As the analysis, we start the detailed explanation of the example with the first statement of `Boot.booted`. Prior to that statement, all pointer targets are unknown. The statement `p = &x` generates a points-to edge $p \to x$. Next is a call to the function `f`. Here, the analysis of `Boot.booted` is interrupted. To determine the effect of `f` on the points-to graph, `f` is analyzed. As `f` does not invoke any other function, intraprocedural analysis suffices. In difference to the analysis in the last section, this time, the boundary information is not that all pointer targets are unknown. At the point where `f` is called, `p` points to `x`, and hence, the analysis of `f` starts with a points-to graph containing that edge. The analysis is straightforward, the only statement in `f` causes the points-to edge $p \to x$ to be replaced by the edge $p \to y$. Thus, at the exit point of `f`, `p` points to `y`. Next, the analysis of `Boot.booted` is resumed with this information. The next statement calls `g`. Therefore, the analysis of `Boot.booted` is interrupted again, the entry node of `g`'s CFG is initialized with $p \to y$, and the analysis of `g` starts. At first, `g` lets `p` and `q` point to `z`. Then, it calls `f`. The function `f` has already been analyzed, but with different boundary information. Therefore, the result from the previous analysis is not reused. The analysis of `g`

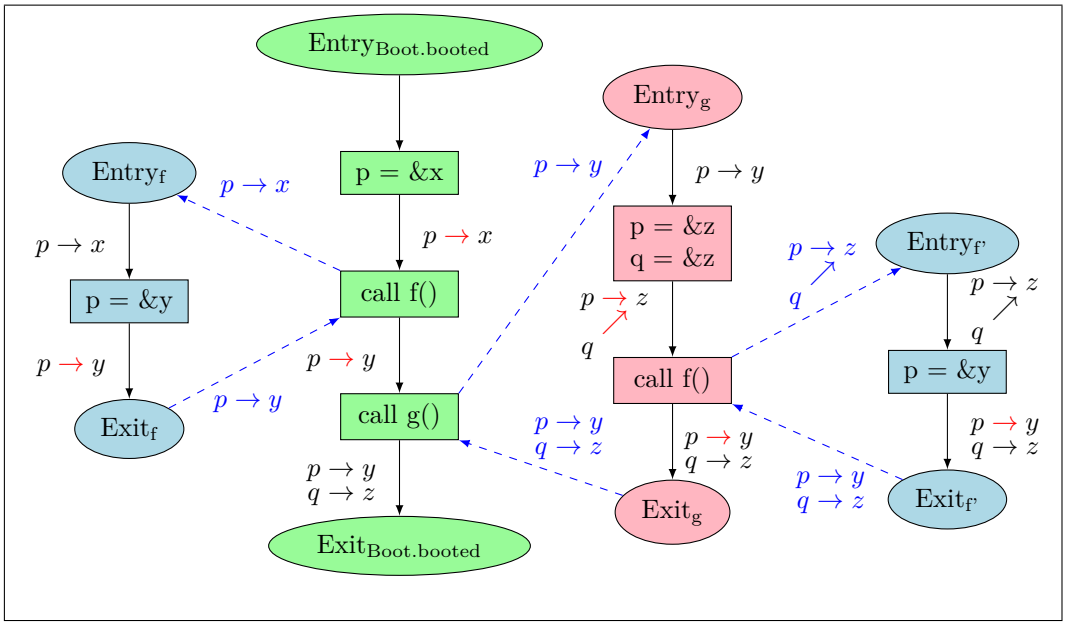

Figure 3.10: Pointer-analysis result for the example program with function calls.

is interrupted and the algorithm analyzes `f` again; this time for $\{p \to z, q \to z\}$, resulting in $\{p \to y, q \to z\}$. This second analysis of `f`, which is independent from the first analysis, is depicted by a separate CFG (f') in the supergraph in figure 3.10. After the result for this invocation of `f` is available, the analysis of `g` is resumed. The call to `f` was the last statement in `g`, therefore, the points-to information at the exit node of `g` is also $\{p \to y, q \to z\}$. Having this result available, the analysis of `Boot.booted` is resumed. The call to `g` was the last statement, and therefore, the points-to information at the exit of `Boot.booted` is $\{p \to y, q \to z\}$.

This example demonstrates the context sensitivity of the analysis: `f` is called from two different program points, with two different points-to graphs. For both invocations, at the end of `f`, the points-to graph contains the edge $p \to y$. However, the edge $q \to z$ exists only when `f` is called from `g`. As `f` is analyzed for both invocations independently, the additional points-to edge $q \to z$ does not reach `Boot.booted` through the direct invocation of `f`, but only through the invocation of `g`.

Depending on the context, we obtain two different results for the points-to graph before the statement in `f`, and also, for the points-to graph after the statement. If a compiler wants to optimize the generated code for `f`, without duplicating the function, it may be interested in all possible points-to relations for `f`, independent from the invocation context. To consider only one context may lead to the wrong assumption that either `p` always points to `x` (f) or `p` always points to `z` (f') at the entry point of `f`. In this case, a summarizing analysis result for `f` is required. For any program point in `f`, a summarizing result can be obtained by combining the

points-to graphs from the distinct analyses at that point. This summarizing result for `f` is shown in figure 3.11.

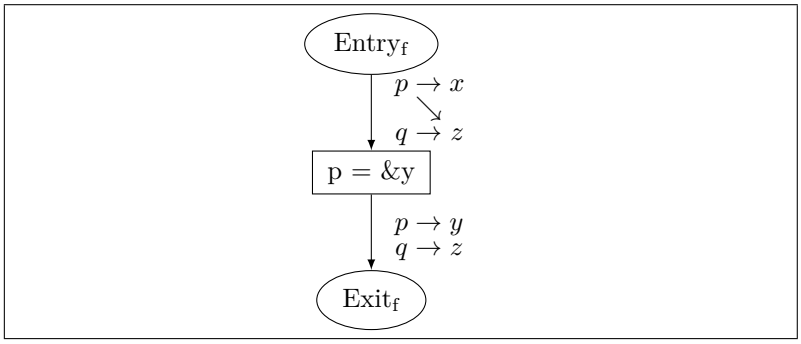

Figure 3.11: The combined pointer-analysis result for function `f` in the example program.

3.3.3 Intermediate Representation

The CFG as intermediate representation of the analyzed program for the intraprocedural analysis was described in section 3.2.3. In this section, the intermediate representation is extended to support function calls. As in the presence of function calls more than one function must be analyzed, there is an individual CFG for every function. Within a CFG, a function call is denoted by a *call* node, as we have seen in the example in the previous section. The call node contains a reference to the invoked function. Therefore, all information to construct a supergraph is available. Nevertheless, a supergraph is never explicitly constructed. The analysis algorithm works with the set of CFGs, and leaves the interprocedural edges implicit. Since a call node contains only a reference and the CFG corresponding to the invoked function is initialized just with the points-to information from the invoking context as boundary information, the analysis behaves as if every call node is effectively replaced with a clone of the called function. This provides context sensitivity. Another difference from a supergraph is, that our intermediate representation has just call nodes. Supergraphs usually split function calls into two nodes, a *call* and a *return* node.

3.3.4 Equations

The new language construct, introduced for the interprocedural analysis, is the function call. In the intermediate representation, a function call is represented by a call node. Thus, the analysis must be extended to handle such call nodes.

Reaching a call node in a CFG means that the effect of the called function on the points-to graph of the caller must be determined. To do so, the analysis of the

calling function is interrupted, to allow the analysis of the called function. This is done in three steps:

1. The points-to information at the call site is mapped into the name space of the called function.
2. The called function is analyzed with this points-to information.
3. The resulting points-to information is mapped back into the name space of the calling function.

For a program that, like the example in figure 3.9, has only global variables and pointers, this mapping is straightforward. All variables and pointers are visible in every function. There are no formal parameters or local variables that are only visible in a specific function, or that hide a global variable or pointer. Therefore, the points-to information at the call site is simply copied to the entry node of the CFG of the called function. Likewise, the points-to information from the exit node of the CFG of the called function is copied back and is propagated as new points-to information from the call node to its successors.

Parameters passed to the called function, values returned to the calling function, or local variables complicate matters. Local variables and formal parameters are only visible in the function they are declared in. Hence, each function may have an individual name space and the algorithm must map the points-to information between those name spaces. All variables that are not visible in the called function are mapped to ghost location sets, to indicate that they are not directly visible, but may be accessed through visible pointers. Using the technique of Rugina and Rinard, the analysis context for the invoked function is set up as follows:

- The location sets for formal parameters and local variables of the calling function are mapped to ghost location sets.
- The location sets for the actual parameters at the call site are mapped to the location sets for the formal parameters of the called function.
- The location sets for local variables of the called function are added to the points-to graph, together with points-to edges from these location sets to `unk`.

The called function is analyzed with this context and the result is mapped back into the naming environment of the calling function as follows:

- The location sets for formal parameters and local variables of the called function are mapped to `unk`.
- The location set for the return value in the called function is mapped to the location for the called function's return value in the calling function.
- The ghost location sets are mapped back to the corresponding location sets for formal parameters and local variables.

After the points-to information is mapped back into the name space of the calling function, the algorithm propagates this information along the outgoing edges of the call node in the control-flow graph, and thereby continues the analysis.

3.3.5 Properties

The interprocedural analysis is an extension to the intraprocedural analysis. Therefore, the properties discussed in section 3.2.5 still hold for the intraprocedural part. Each called function is analyzed with the specific points-to information from the call site as boundary information and the resulting information is transferred back to the specific call site. Thus, the interprocedural analysis is context sensitive. Since the intraprocedural analysis is sound and the interprocedural analysis uses the intraprocedural analysis to compute a safe approximation of the effect of a called function on the points-to information, the interprocedural analysis is also sound. Unfortunately, completely reanalyzing a function in every calling context highly impacts the complexity. While the intraprocedural analysis has a polynomial upper bound for the runtime, the interprocedural analysis has an exponential upper bound, a property that it shares with other context-sensitive analyses.

Comparing our algorithm to Rugina and Rinard's algorithm, we have two additional restrictions: our analysis does not support function calls through function pointers or recursive function calls. The algorithm does not terminate for recursive function calls and invocations through function pointers are not handled at all. In the context of TinyOS applications, this is not a real restriction. First, function pointers are unnecessary due to nesC's component and interface model, that uses wiring to statically replace function pointers. Second, according to the TinyOS programming hints, recursive function calls should be avoided.

3.4 Event-Sensitive Analysis

In the previous section, the intra- and interprocedural part of the analysis algorithm were described. This section introduces a novel approach to analyzing event-driven programs. These programs are represented by *event graphs* that model the possible execution orders of the events. Concurrently executing events are modeled as concurrent threads. It is thus possible to use the algorithm for multithreaded programs for the analysis of event-driven programs. To express event-driven programs, the input language is extended to support events. Thereafter, the event graph is introduced as intermediate representation and the analysis algorithm is presented in its final form.

NesC components interact with each other through interfaces. Interfaces consist of *commands* and *events*. Commands are implemented by the component providing the interface, and are used by a component using the interface to request a service from

the component providing the interface. Likewise, events are implemented by the component using the interface, and are used by a component providing the interface to inform the interface user that something happened. We restrict our analysis to a single nesC component, more precisely to the module that implements the top-level application logic. Therefore, no other component will request a service from the analyzed module, and thus, commands are not considered in the analysis.

3.4.1 Language

The input language is an extension to the language specified in section 3.2.1 and section 3.3.1. The new constructs are *events*, *tasks*, and *command calls*, which trigger the future execution of events. The syntax follows the syntax of nesC. An event is essentially a function declared with the keyword `event`. Like functions, events may take parameters from the caller and return a result. An event `e` that takes no parameters and returns no value is declared as `event void e()`. The characteristic of events is, that events are invoked from somewhere outside the declaring module. Therefore, from the modules perspective, it is unknown at what time the event is executed.

Commands, like normal functions, are invoked by their name and a list of parameters in parentheses, except that the keyword `call` is written before the name. A command `c` that takes no parameters is called by `call c()`. As said above, we restrict the analysis to the top-level module of the application. Commands are used by other components to request a service, but the top-level module does not offer services to other components. Thus, there is no need for the input language to support commands. Commands behave like normal functions, but called commands belong to a different component. Therefore, they are beyond the scope of the analysis. As called commands are not analyzed, it is only possible to call commands that have no effect on the points-to relations in the analyzed module. Calls to *split-phase* commands are important to the analysis, and the reason why command calls are actually included in the input language. Even if the command itself must not alter the points-to relations, it will eventually invoke an associated event of the caller. The invoked event is a part of the analyzed module, and therefore may influence the points-to relations.

Another construct of the input language, and nesC, is tasks. Like events, tasks are essentially functions, and are declared with the `task` keyword. Unlike events, a task is only accessible from the module it is declared in. A task is not invoked directly, but *posted* to the task queue, from where the scheduler will fetch and execute it later. Thus, tasks enable the deferred execution of functions. The scheduler is free to decide when to execute the task. Therefore, similar to events, it is unpredictable when the task is invoked.

3.4.2 Example

```
  uint8_t x, y ,z;
  uint8_t *p;

  event void Boot.booted() {
1:  p = &x;

2:  call RadioCtl.start();
3:  call Timer.startOneShot(...);

4:  p = &z;
  }

  event void RadioCtl.startDone(...) {
5:  p = &x;

6:  call Radio.send(...);
  }

  event void Timer.fired() {
7:  p = &y;
  }

  event void Radio.sendDone(...) {
8:  p = &z;
  }
```

Figure 3.12: A program to demonstrate the event-sensitive analysis.

In the previous section, the input language was extended to its final form. To illustrate that extension, this section discusses an event-driven program that demonstrates the event-sensitive analysis, before the required extension of the analysis itself is formally introduced in the next sections. The essential part of the considered application is given in figure 3.12. For simplicity, some details, including interface declarations and parameters for command calls and events, have been omitted. The application consists of three 8 bit wide unsigned integer variables (`x`, `y`, `z`), one pointer (`p`), and four events (`Boot.booted`, `RadioCtl.startDone`, `Timer.fired`, `Radio.sendDone`). In short, the application starts the radio, once the radio is ready it transmits a message and eventually receives a notification that the message was sent. Furthermore, the application starts a timer that eventually invokes the `Timer.fired` event. This timer event has the special characteristic that it is unknown, when, in relation to the radio events, it is executed. We are not interested in the details of sending messages, but in the effects of concurrently executed events on the points-to relations. The *event graph* in figure 3.13 depicts the application's events and models their relationship.

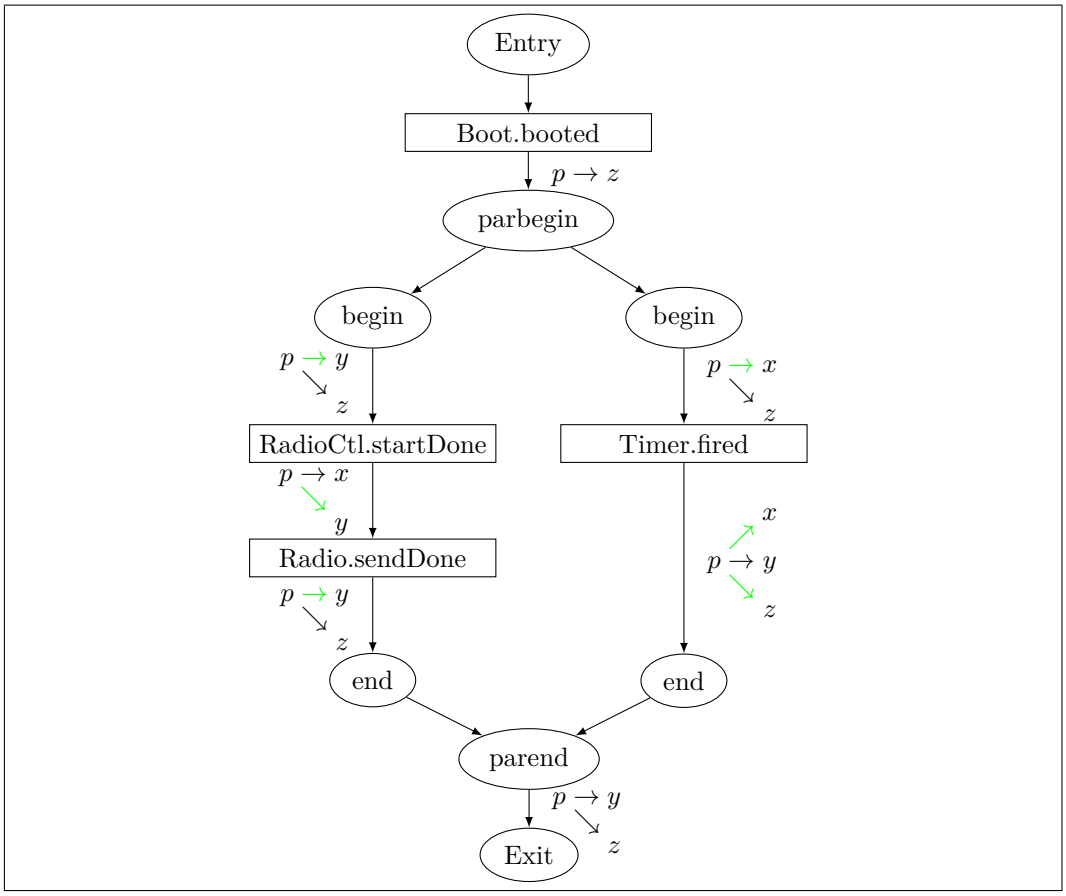

Figure 3.13: Pointer-analysis result for the example program with events. Black arrows indicate points-to relations generated in the same thread, green arrows interfering points-to relations generated in a concurrently executing thread.

We will now look into the example in more detail, starting with the event graph itself. The event graph models the relationship between the individual events. The applications control flow is considered as a thread of events that may start other concurrently executing threads of events. From the TinyOS specification it is known that `Boot.booted` is the first event of the application that is invoked. The assignments in `Boot.booted` are irrelevant for the construction of the event graph, but the two command calls are split-phase, and thus lead to the future execution of events of the analyzed module. After executing the `RadioCtl.start` command, the radio will eventually signal its availability by invoking the `RadioCtl.startDone` event. Likewise, the call to the `Timer.startOneShot` command will eventually lead to the invocation of the `Timer.fired` event. The analysis algorithm can not determine which event will be executed first. Therefore, it must consider both execution orders: `RadioCtl.startDone` before `Timer.fired` and `Timer.fired` before

`RadioCtl.startDone`. This indefinite execution order is expressed by two threads, executing in parallel. The begin of a parallel execution is indicated by a "parbegin" node in the event graph. Each of the individual threads starts with a "begin" node. `RadioCtl.startDone` is the first event in the left thread. During its execution, it calls the `Radio.send` command, leading to the future invocation of the `Radio.sendDone` event. Therefore, `Radio.sendDone` must be executed after `RadioCtl.startDone`, and thus, is the second event in the left thread. It contains no statement that triggers the future execution of another event. Hence, the left thread of events ends. This is indicated by the "end" node in the event graph. The right thread consists only of the `Timer.fired` event, which is followed by that thread's end node. A "parend" node indicates the end of the parallel execution, and finally, the last node in the event graph is an exit node that indicates the end of the application's main thread.

Once the event graph is available, the analysis algorithm starts to analyze the application. It starts at the entry node on top of the graph, with the boundary information that the targets of all pointers are unknown. Following the control flow of the main thread, the first event is `Boot.booted`. An *event node* in the event graph is similar to a call node in a CFG. To determine the effect of `Boot.booted` on the points-to graph, `Boot.booted` is analyzed with the intra- and, if required, interprocedural analysis. The statement at line 1 leads to a points-to edge $p \to x$. The next statements, at line 2 and line 3, are command call statements. They were used to construct the event graph, but have no immediate effect on the points-to graph, as they are implemented in another module, and therefore, not in the scope of this analysis. The statement at line 4 makes `z` the new target of `p`, leading to the points-to edge $p \to z$ at the exit of `Boot.booted`. This resulting points-to graph is depicted after the node for the `Boot.booted` event in the event graph. The two parallel threads are analyzed next. We start with the left one, and in the first pass, ignore the points-to edges drawn in green. Coming from `Boot.booted`, $\{p \to z\}$ is passed as boundary information to the analysis of `RadioCtl.startDone`. `RadioCtl.startDone`, at line 5, kills the edge $p \to z$ and generates the edge $p \to x$. Then, `Radio.sendDone` kills $p \to x$ and generates $p \to z$ at line 8. Thus, the points-to information reaching the end node of the left thread is $p \to z$. We continue the analysis with the thread on the right-hand side. It consists of a single event: `Timer.fired`. This event may be executed before `RadioCtl.startDone`. In this case, the boundary information for the analysis of `Timer.fired` is $\{p \to z\}$, the points-to relations after the execution of `Boot.booted`. It can also be executed after `RadioCtl.startDone` and before `Radio.sendDone`. Then the boundary information for the analysis of `Timer.fired` is $\{p \to x\}$. Finally, it may be executed after `Radio.sendDone`. The boundary information is then $\{p \to z\}$, coincidentally the same as when `Timer.fired` is executed directly after `Boot.booted`. Hence, the undetermined execution order leads to an additional possible points-to relation $p \to x$ at the beginning of the execution of `Timer.fired`. In figure 3.13, these additional points-to edges are depicted in green color. Directly after the execution of `Timer.fired`, `p` definitively points to `y`.

However, a moment later, one of the events from the left thread may execute and override the assignment from `Timer.fired`. Therefore, after `Timer.fired` executed and before both threads finished their execution, `p` may point to `x`, `y`, or `z`. After the points-to information from the right thread is available, the left thread is reanalyzed, to incorporate the points-to edge $p \rightarrow y$, generated in `Timer.fired`. The points-to information of the two analyzed threads reaches the "parend" node, at which it is merged in a way to reflect the points-to relations after both threads finished their execution. That is, either `Timer.fired` executed last, leading to the points-to relation $p \rightarrow y$, or `Radio.sendDone` executed last, leading to the points-to relation $p \rightarrow z$. Therefore, the points-to graph contains both of these edges. `RadioCtl.startDone` triggers the future execution of `Radio.sendDone`, so the points-to relation $p \rightarrow x$ is always overridden before that thread of events ends. Consequently, the points-to graph after the parend node does not contain an edge $p \rightarrow x$.

It is fundamental to note that the points-to relation $p \rightarrow x$, valid at the time `RadioCtl.start` was called, has no influence on the boundary information for the `RadioCtl.startDone` event. This independence of points-to information from command call and event execution is a significant difference to normal function calls. A normal function call passes the points-to relations to the called function and that function is executed immediately. A command call also passes the points-to relations to the called command, but the associated event is invoked later, observing the effective points-to relations at the time of its execution.

3.4.3 Intermediate Representation

This section completes the description of the intermediate program representation for the presented pointer-analysis algorithm. Section 3.2.3 introduced the CFG as intermediate representation for functions consisting of basic assignment statements and of control-flow statements. In Section 3.3.3, call nodes were added to the CFG to represent function calls. This is sufficient to represent all individual functions and events, and allows modeling the invocation of one function from another. However, such a CFG, in general, can not model all possible execution orders of the application's events. Therefore, the *event graph* is introduced as a new intermediate representation, to describe the program at the event level.

At first glance, an event graph is similar to a CFG. Figure 3.13 of the example in section 3.4.2 depicts an event graph. As a CFG, it has an entry node, indicating where the execution starts, and an exit node, indicating where the execution ends. Furthermore, the edges in both graphs represent the control flow. However, a node in a CFG represents a single statement or block, whereas a node in an event graph represents an entire event. In addition, an event graph may contain nodes labeled "begin" or "end", denoting that the nodes in between form a *thread of events*. Finally, "parbegin" and "parend" nodes are used to indicate that the threads of events between a parbegin and the corresponding parend node are executed concurrently.

This enables the event graph to represent the non-deterministic execution order of events.

In an event graph, each event is represented by one or more *event nodes*. Additionally, every function and event is represented by a CFG, as described for the intra- and interprocedural analysis. An event node is similar to a call node in a CFG; it contains a reference to the invoked event. When the analysis algorithm reaches an event node, it looks up the CFG corresponding to the event. Then it copies the points-to information, reaching the event node, as boundary information to the entry node of the event's CFG and starts to analyze the event. After the analysis of the event is finished, the resulting points-to information is transferred back from the CFG's exit node to the event graph and is propagated to the next node. This is similar to the procedure for function calls.

The first event that is invoked is `Boot.booted`. Therefore, its corresponding event node is the successor of the entry node. Whenever an event e triggers the future execution of another event, an event node for that other event is added to the event graph as a direct successor to the node corresponding to the event e. The interesting part is when an event triggers the future execution of more than one event. For some events, like timer events, it may be possible to determine the execution order of these triggered events. However, in general, the order is non-deterministic. To represent all possible execution orders, a new thread of events is created for each of these triggered events, with the triggered event at the beginning. The event nodes of a thread of events are enclosed in "begin" and "end" nodes for the thread. Therefore, the node for the triggered event is added as a direct successor to the begin node. A pair of "parbegin" and "parend" nodes indicates that the enclosed threads are executed in parallel. Thus, a parbegin node is added as direct successor to the node for the event e and the begin nodes of the individual threads are added as direct successors to the parbegin node. The triggered events themselves may invoke events that are consequently added to the invoking event's thread. If an event does not trigger the future execution of any other event, the thread it belongs to ends. Therefore, the thread's end node is the direct successor of this event's node. The end node itself has the parend node as its only direct successor. Finally, the exit node is added to the event graph, such that it postdominates all other nodes. The example in section 3.4.2 illustrated the event graph construction.

Another concept to model is that of recurring events. Recurring events are events that are signaled more than once during the program execution. If it is guaranteed that an event only recurs after all of the other events that its execution triggered have finished their execution, the recurrence can be modeled by a backedge from the end of this chain of events to its beginning, as long as the constraints for event graphs are not violated. Examples of such events are timer events with intervals much longer than the time it takes to execute the event handler and the related code. Events signaling incoming network messages, on the other hand, may arrive before the processing of the previous message is completed. Therefore, this type

of recurring events can not be modeled with backedges, as all possibly interleaved executions of these chains of events must be considered. The nodes for such events are placed between "recbegin" and "recend" nodes in the event graph. The nodes "recbegin" and "recend" are similar to "parbegin" and "parend" nodes respectively. The difference is that the threads in the par-construct are different from each other and there is exactly one instance of each thread, while the rec-construct contains just one thread, but models the parallel execution of infinitely many instances of it.

There are some constraints regarding the control flow through threads and their enclosing parbegin and parend nodes as well as recbegin and recend nodes. There are no edges allowed between any nodes belonging to different parallel threads. Therefore, every node of a thread is dominated by the thread's begin node and postdominated by the thread's end node. Each node of a thread, except the begin node, has all incoming edges only from nodes of the same thread. Likewise, each node of a thread, except the end node, has all outgoing edges only to nodes of the same thread. For each thread, there is exactly one edge from the corresponding parbegin node to the thread's begin node and exactly one edge from the thread's end node to the corresponding parend node. There are no other edges to begin or parend nodes and no other edges from end or parbegin nodes. The same constraints apply to nodes for threads of recurring events. Every event node in a chain of recurring events is dominated by the recbegin node and postdominated by the recend node.

Besides events, the input language supports tasks. Tasks are functions in the same module whose execution is deferred. A task is *posted*, i.e., marked for future execution, and later selected and invoked by the scheduler. Therefore, tasks can be treated like events in the event graph. Tasks are also represented by event nodes, and whenever a task is posted, an event node corresponding to the task is added as successor of the task or event posting that task. Hence, we will not differentiate between tasks and events.

3.4.4 Equations

The event graph was introduced in the previous section. This section presents the data-flow equations to analyze programs represented by event graphs. Event nodes are equal to call nodes in a CFG. One might therefore expect that only threads and groups of parallel threads require additional data-flow equations. Unfortunately, this is not entirely true. In order to handle parallel threads, we must revise and extend the previously presented equations, starting with the definition of points-to information.

Definition 3.1 on page 22 defined the points-to information at a program point p as the points-to graph at p. To capture the effect of assignments in concurrently executing threads, the definition of points-to information is extended to *parallel points-to information* as follows:

Definition 3.2. Let L be the set of all location sets in the program and $P = 2^{L \times L}$ the set of all points-to graphs for these location sets. The *parallel points-to information* PPI(p) at a point p in the program is a triple $\langle C, I, E \rangle \in P^3$ consisting of:

- C: the current points-to graph,
- I: the set of interference edges generated by concurrently executing threads,
- E: the set of all points-to edges generated in the current thread.

The current points-to graph, C, as before, represents the points-to relations at a program point p. The *interference graph*, I, describes the effect that pointer assignments in other threads have on the points-to relations in the currently analyzed thread. It consists of all points-to edges generated in concurrently executing threads. The set of all points-to edges generated in the current thread, E, is used to compute the interference graph for the other threads, executing in parallel.

Extending the points-to information from a single points-to graph to a triple of these graphs makes it necessary to revise the ordering, and therefore, the lattice of points-to information. As the parallel points-to information is a triple of points-to graphs, we will consider the product lattice. The partial order from P is extended to P^3 by considering the partial ordering of the individual sets of points-to edges:

$$\langle C_1, I_1, E_1 \rangle \sqsubseteq \langle C_2, I_2, E_2 \rangle \iff C_1 \subseteq C_2 \wedge I_1 \subseteq I_2 \wedge E_1 \subseteq E_2$$

Therefore, $(P^3, \sqsubseteq)$ is a poset and forms a lattice with the extended meet operator:

$$\langle C_1, I_1, E_1 \rangle \sqcup \langle C_2, I_2, E_2 \rangle := \langle C_1 \cup C_2, I_1 \cup I_2, E_1 \cup E_2 \rangle$$

Next, we have to redefine the functional $[\![\ ]\!]$ that assigns a transfer function to every statement of the program. *Stat* is still the set of all statements in the program, but the extension of the points-to information to PPI(p) $\in P^3$ must be reflected in the transfer function f, which is now $f \in P^3 \to P^3$. Therefore, the functional $[\![\ ]\!]$ becomes $[\![\ ]\!] : \mathit{Stat} \to (P^3 \to P^3)$. For basic assignment statements, this extended functional is shown in figure 3.14 and, as before, defined in terms of *gen* sets, *kill* sets, and a *strong* flag. The definitions of the *gen* set, the *kill* set, and the *strong* flag are the same as for the earlier analyses, given in figure 3.8 on page 23.

The extensions are straightforward. The interference edges created by all assignments in the other, concurrently executing threads are also included in the current points-to graph. If the set of interference edges is empty, the algorithm computes the same current points-to graph as the analysis for sequential programs, presented in section 3.2. An assignment in the current thread has no effect on the interference edges generated by other threads. Therefore, the interference graph is unchanged. Finally, as E is the set of all points-to edges generated in the current thread, all edges in the *gen* set for the current statement must be included in E.

$$[\![st]\!]\langle C, I, E\rangle = \langle C', I', E'\rangle, \text{ where:}$$

$$C' = \begin{cases} (C - \mathit{kill}) \cup \mathit{gen} \cup I & \text{if } \mathit{strong} \\ C \cup \mathit{gen} \cup I & \text{if not } \mathit{strong} \end{cases}$$
$$I' = I$$
$$E' = E \cup \mathit{gen}$$

Figure 3.14: Data-flow equations for the event-sensitive analysis.

Having these extensions to the intraprocedural analysis in place, the extension of the interprocedural analysis is straightforward. The two additional sets, I and E, must also be mapped into the scope of the called function, like the set C, before the analysis of that function starts, and afterwards must be mapped back into the scope of the caller. Next, we consider the analysis of parallel threads. The data-flow equations for the parallel construct and threads are given in figure 3.15.

$$[\![\texttt{par}\{\{t_1\} \ldots \{t_n\}\}]\!]\langle C, I, E\rangle = \langle C', I, E'\rangle, \text{ where:}$$

$$C' = \bigcap_{1 \le i \le n} C'_i$$
$$E' = E \cup \bigcup_{1 \le i \le n} E_i$$
$$C_i = C \cup \bigcup_{\substack{1 \le j \le n \\ j \ne i}} E_j$$
$$I_i = I \cup \bigcup_{\substack{1 \le j \le n \\ j \ne i}} E_j$$

$$[\![t_i]\!]\langle C_i, I_i, \varnothing\rangle = \langle C'_i, I_i, E_i\rangle$$

Figure 3.15: Data-flow equations for the parallel construct and threads.

The parallel points-to information flowing into a parbegin node passes through it unchanged and is propagated to the begin nodes of all threads. At a thread's begin node, the individual points-to information for the thread is computed. The thread may be the first one that is executed. Therefore, all edges of the current points-to graph C reaching the parbegin node, are included in the current points-to graph C_i at the begin of the thread. Furthermore, any points-to relation that is generated by the other threads may be valid at the time the thread's execution starts. Thus,

all points-to edges generated in the other threads are also included in the current points-to graph C_i. Interference edges describe the points-to relations generated by threads executing in parallel to the current one. As threads executing in parallel to the entire `par` block also execute in parallel to the individual threads of that block, all interference edges reaching the parbegin node are included in the set of interference edges I_i for an individual thread. As for the current points-to graph, all points-to edges generated by other threads of the same `par` block are also interference edges for the current thread. The E set just accumulates the edges generated by the current thread, and hence, it is empty at the begin of the thread. The current points-to graph C_i' at the end of a thread is the same as the one reaching the end node. The same holds for the set E_i of all generated points-to edges for that thread. Interference edges are only computed at the threads begin node, and then left unchanged inside the thread. Therefore, the set of interference edges I_i at the begin of the thread is also the set of interference edges at the end of the thread. At the end of the parallel block, i.e., when all threads have finished their execution, the points-to information from the different threads must be combined. The interesting part is the resulting current points-to graph C'. It must contain all edges that reached the parbegin node and were not killed by a strong update in any thread. These points-to edges are in the points-to graphs at the exit of all threads, and therefore in the intersection of the sets C_i'. In addition, the graph must include all points-to edges generated in any thread that are not killed later in the same thread. These edges get into the points-to graphs of the other threads as interference edges, and thus, are also in the intersection of the sets C_i'. The set of interference edges is updated only at the begin node of a thread. Therefore, the set of interference edges at the parend node is the same as it was at the parbegin node. Finally, at the parend node, the set E' of all points-to edges generated in the current thread, must include all edges it contained at the parbegin node and all edges generated in all threads of the parallel thread group.

Chains of recurring events execute in parallel and that leads to interleaved executions of the related events. Therefore, these event chains are treated as threads that execute in parallel. Since, in this case, all threads consist of the same sequence of events, it is sufficient to analyze just one thread instance. The interference information is computed from this instance and then merged into the points-to graphs for the thread. Figure 3.16 lists the data-flow equations for recurring events. These equations yield a safe approximation of the points-to relations for an unknown number of concurrently executing instances of event chains. There is a difference in the possible points-to relations between one and two instances executing in parallel, but any number of instances higher than two yields the same possible relations as two instances. Furthermore, the possible points-to relations for one instance are a subset of the relations for more than one instances and thus, these equations yield a sound result for any number of instances executing concurrently.

The extension of the points-to information from a single graph to a triple of such graphs also influences the initialization values. The algorithm presented in sec-

$$[\![\texttt{rec}\{c\}]\!]\langle C, I, E\rangle = \langle C'_c, I, E'\rangle, \text{ where:}$$

$$E' = E \cup E_c$$
$$C_c = C \cup E_c$$
$$I_c = I \cup E_c$$

$$[\![c]\!]\langle C_c, I_c, \varnothing\rangle = \langle C'_c, I_c, E_c\rangle$$

Figure 3.16: Data-flow equations for recurring events.

tion 3.2 initialized all CFG nodes, except the entry node, with an empty set of points-to edges. This initialization is extended: the event-sensitive algorithm initializes all event graph and CFG nodes, except the entry node of the event graph, with a triple $\langle \varnothing, \varnothing, \varnothing \rangle$ consisting of empty sets. At the entry node of the event graph, i.e., before any statement of the analyzed application was executed, the current points-to information is, as before, that all pointers point to unknown locations. The main thread is not executed in parallel with any other thread. Therefore, the set of interference edges is empty. Without any statement executed, there can be no generated points-to edges, and thus, the set of all edges generated in the current set is also empty. Hence, the boundary information at the entry node of the event graph is $\langle L \times \{\texttt{unk}\}, \varnothing, \varnothing \rangle$.

3.4.5 Properties

This section concludes the presentation of the pointer-analysis algorithm for event-driven programs by discussing its properties. There are two essential extensions to the analysis that was presented in sections 3.2 and 3.3: the event graph to model the dependencies of events, and the possibility to concurrently execute code. As long as the order of execution is deterministic for all events, the event graph describes this sequence. For the event-sensitive analysis, the points-to information is propagated along the edges of the event graph as it is propagated along the edges of the CFG for the intraprocedural analysis. The event nodes are like call nodes in the interprocedural analysis. Therefore, the event-sensitive analysis, in the strictly sequential case, is a straightforward extension of the intra- and interprocedural analyses, and thus, it is sound.

The other extension is the concurrent execution of events, modeled by the par- and rec-constructs. One thread of events influences parallel threads by creating or killing points-to relations between shared pointers and variables. Since a safe approximation must have a points-to edge $p \to x$ whenever `p` contains the address of `x`, the analysis must consider the points-to edges generated by events in parallel threads, but may neglect killed ones. Therefore, all points-to edges generated by

a thread are captured by the E set and merged into the interference graph I of the other threads. The data-flow equations ensure that all edges in the interference graph at a program point are also in the current points-to graph at that program point. Thus, the analysis is sound for parallel threads of events. The rec-construct for recurring events is a special case of the par-construct. It models all possible interleaved executions of an instance of a thread with other instances of the same thread. As for the par-construct, the computed interference graph ensures that all points-to edges possibly generated by another instance are present in the current points-to graph. Hence, the analysis is sound.

3.5 Integration into the NesC Toolchain

The algorithm presented in the previous sections is implemented in the nesC compiler. The nesC compiler is a source-to-source compiler that translates nesC code to C code. This C code is later compiled to machine code by a C compiler. The nesC compiler parses the application's source code and connects the individual components (wiring). After that, the *nesC graph extractor* (NGE), an experimental compiler extension by Arne Wichmann and Volker Menrad that is still under development, creates the *megagraph* for the application's top-level module. A megagraph is an extended supergraph for event-based systems. In contrast to the presented event graph, the megagraph represents every event exactly once. If an event triggers the future execution of another event, there is an *invoking edge* from the exit node of the triggering event to the entry node of the triggered event. The nondeterministic execution order is represented by additional *interleaving edges*. There is an interleaving edge from the exit node of one event to the entry node of another event, if the execution of the other event is not triggered by the first event, but may nevertheless happen after the first event finished. This does not fit in the model of the event graph, which requires that concurrently executing events are modeled as a group of parallel threads and that there are no control-flow edges between any nodes belonging to different threads. Therefore, the implementation of the presented algorithm does not use the megagraph, but it uses the information it provides, namely what event triggers the future execution of what other event. The available NGE implementation can not infer the relation between a command call and the future execution of an event. The user of the compiler must provide these split-phase relations in a separate file. Based on the information from the megagraph, the implementation of the points-to analysis creates the event graph for the application. For testing purposes and to cover applications for which the current implementation fails to create a correct event graph, the user of the compiler may also provide a file containing an event-graph specification.

The main components of the nesC compiler, the relevant extensions, and the exchanged data-structures are depicted in figure 3.17. On the left-hand side are the input files, namely the nesC source code, the split-phase information that relates

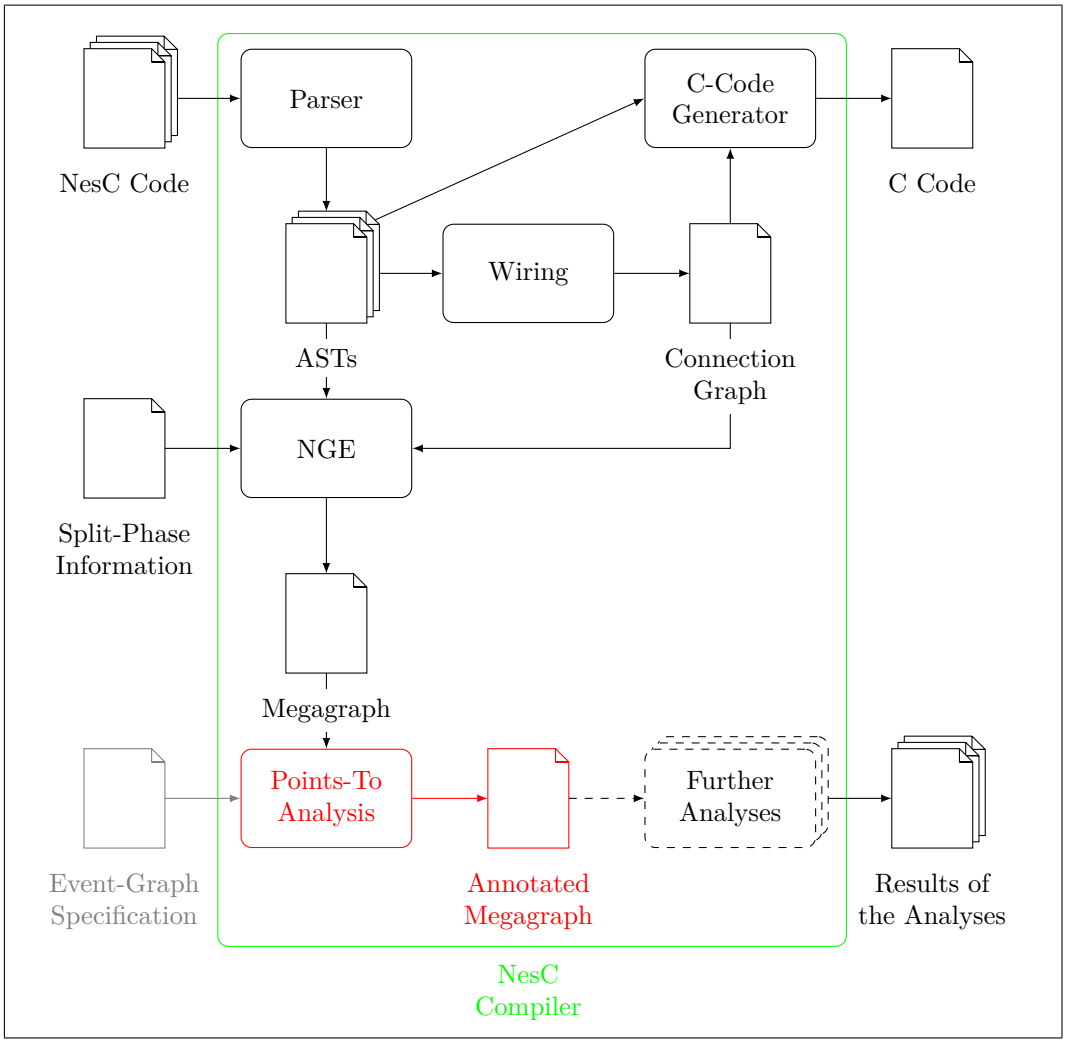

Figure 3.17: Schematic depiction of the nesC compiler with NGE and points-to analysis extensions.

commands to events, and the optional event-graph specification. On the right-hand side are the output files, namely the application as a single file of C code and the results of the implemented program analyses. At the moment, the presented points-to analysis (depicted in red) is the only implemented program analysis. However, its result, the megagraph annotated with possible points-to relations (also depicted in red), will be the input to other analyses currently under development. One of these planned analyses is a *memory ownership analysis* that determines what component of an application owns what memory location, to enforce TinyOS programming hint 4: only one component should be able to modify a pointer's data at any time [20].

An event is represented by a single node in the event graph. This is not enough to analyze the event's effects on the points-to relations. Therefore, a CFG is required for every event and function that is analyzed. The megagraph, as an extended supergraph, contains the required CFGs as subgraphs. Thus, the intraprocedural part of the implemented points-to analysis operates on the available megagraph. Since the megagraph contains nodes and edges that belong to events and functions other than the analyzed one, it is important to limit the intraprocedural analysis to the currently analyzed function. Hence, it must start at the entry node of the analyzed function and ensure that it does not follow any control-flow edges leaving the exit node. In contrast to the event graph and the CFGs presented in this work, the megagraph has the statements on its edges. Nodes only indicate the begin and end of a function, separate the edges of two adjacent statements, and are used to split and merge the control-flow. For simplicity, the points-to analysis implementation follows this design for the intraprocedural analysis. The implemented event graph still has the events as its nodes.

The analysis starts by creating location sets for the module's global variables. Then it creates the event graph from the information available in the megagraph and attaches the initial points-to information to the nodes and edges. The implementation uses a worklist to record the nodes that must be analyzed. It adds the event graph's entry node to the worklist and enters the analysis loop. In the loop, a node is fetched from the worklist. Then the meet of the points-to information from the incoming edges is computed and compared to the points-to information stored at the node. If there are differences, the stored points-to information is updated and the node is analyzed. For an event node, the algorithm performs an interprocedural analysis of the corresponding event. For a begin node of a thread, the algorithm computes the C_i and I_i sets to propagate them into the thread. For a parend node, C' and E' are computed and stored as outgoing points-to information. Finally, for end or parbegin nodes, the points-to information is simply copied from the in-set to the out-set. If the points-to information stored at the node was changed, the successors of the node are added to the worklist. For an end node of a thread, the begin nodes of all other threads of its group are added to the worklist additionally. The algorithm terminates when the worklist is empty.

The implementation of the intraprocedural analysis is similar. It initializes the subgraph of the megagraph that forms the analyzed function's CFG. Thereby it replaces the data stored at the megagraph's nodes and edges with the initial points-to information. After the analysis of the function is completed, the original data is restored and the computed points-to relations are added to it. The implementation of the intraprocedural analysis, like the analysis on the event graph level, uses a worklist to record nodes that must be analyzed. The difference here is that the nodes do not correspondent to statements. Therefore, only the meet of the incoming points-to information is computed and, if it is different from the information stored previously, the statements at the outgoing edges are analyzed with this new information. Then

the successors of the analyzed node are added to the worklist, to analyze them with the newly computed information.

The pointer analysis is enabled together with the megagraph construction by invoking the compiler with the `-fnesc-nge-meta` flag. The megagraph annotated with the possible points-to relations at every program point is written to a file in DOT format, if the file is specified by the `-fnesc-dfa-pa-graphfile` flag. For further debugging purposes, the CFGs resulting from the intraprocedural analyses can also be written to files by invoking the compiler with the `-fnesc-dfa-pa-write-debug-files` flag. The `-fnesc-dfa-pa-eventgraph` flag allows to specify a file that contains an alternative event-graph layout. If specified, that layout overrides the automatically constructed event-graph. Finally, the `-fnesc-verbose` flag lets the compiler write additional information to the standard error stream. The points-to analysis contributes to this additional information by reporting, which statement is currently analyzed, what the computed *gen* and *kill* sets are, and which nodes are added to the worklist.

4 Evaluation

The points-to analysis algorithm for event-driven programs was presented in the previous chapter. A large part of the analysis has been implemented in the nesC compiler and was used to evaluate the algorithm. This chapter presents the findings. Section 4.1 describes the test setup. Sections 4.2 to 4.5 evaluate the obtained analysis results for programs that test the correct handling of the language constructs introduced in chapter 3. Finally, section 4.6 presents the results of the evaluation.

4.1 Test Setup

Since the analysis is implemented in the nesC compiler and runs as a step of the compilation process, the test applications were compiled with the nesC compiler. The points-to analysis is enabled by invoking the compiler with the `-fnesc-nge-meta` flag. This flag also enables the construction of the megagraph that contains the CFGs that are required for the points-to analysis. The module that should be analyzed is specified by the `-fnesc-nge-components` compiler flag. Furthermore, to evaluate the analysis process, the compiler is invoked with `-fnesc-verbose` to write a trace to the console, and with `-fnesc-dfa-pa-write-debug-files` to produce CFGs annotated with points-to information for every analyzed function. Since the applications use the build system of TinyOS the compilation is run with:

```
CFLAGS="-fnesc-nge-components=MODULE\
 -fnesc-nge-meta\
 -fnesc-verbose\
 -fnesc-dfa-pa-write-debug-files\
 -fnesc-nge-help=nge-help.txt\
 -fnesc-nge-invokeout=invoke.dot\
 -fnesc-nge-metaout=meta.dot"

make CFLAGS="${CFLAGS}" iris
```

`MODULE` is a placeholder for the name of the module to analyze. The additional flags that were not explained above are required by the megagraph-construction code. The file given with the flag `-fnesc-nge-help` specifies split-phase pairs of commands and events. Each line of the file contains a command name, an event name, and a number

that indicates how often the event is triggered by a single call to the corresponding command. For example, calling the command `Timer.startOneShot()` leads to a single `Timer.fired()` event, but calling `Timer.startPeriodic()` leads to recurring `Timer.fired()` events. The last two compiler flags specify files where the NGE compiler extension writes intermediate graphs into. This is not necessary for the points-to analysis, but the NGE extension requires these files to be specified. The target platform is irrelevant to the analysis of the top-level module, and therefore "iris" was chosen arbitrarily.

4.2 Basic Assignment Statements

The intraprocedural analysis considers the effects of basic assignment statements. Therefore, the evaluation starts with the analysis of an application that consists of basic assignment statements. We use a program similar to the module given in figure 3.3 on page 18. For reasons of space, the second constant assignment, `y = 0`, has been removed from the program.

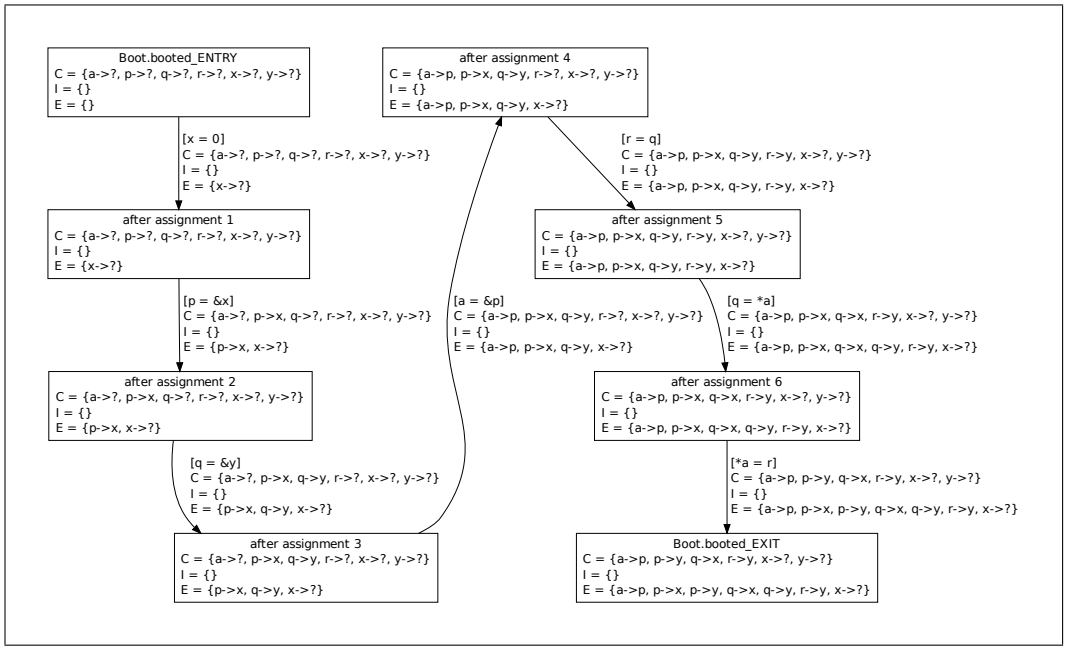

Figure 4.1: CFG for the program to evaluate the analysis of basic assignment statements, annotated with parallel points-to information.

The compiler-generated CFG for the event `Boot.booted` of the analyzed program, annotated with the computed points-to relations, is shown in figure 4.1. As mentioned in section 3.5, in this CFG, statements are represented by edges rather than nodes. The trace on the console (not depicted) shows that the following location sets

were identified: $\langle ?, 0, 0\rangle$, $\langle a, 0, 0\rangle$, $\langle p, 0, 0\rangle$, $\langle q, 0, 0\rangle$, $\langle r, 0, 0\rangle$, $\langle x, 0, 0\rangle$, and $\langle y, 0, 0\rangle$. $\langle ?, 0, 0\rangle$ is the `unk` set that is automatically generated for every analysis and the other six location sets correspond to the six module variables. Therefore, the location sets were identified correctly. The set C of the boundary information at the "Boot.booted_ENTRY" node contains six points-to edges, one from each of the location sets for the module variables to `unk`. The sets I and E at that node are empty. Thus, the implemented algorithm generated the correct boundary information.

The control-flow edge from "Boot.booted_ENTRY" to "after assignment 1" represents the constant assignment `x = 0`. The trace on the console shows the computed *gen* set, *kill* set, and *strong* flag. The *kill* set contains the points-to edge $x \to ?$ that reached the assignment statement, the *gen* set also contains the edge as $x \to ?$, and the update is strong. According to the rules for constant assignments, given in figure 3.8 on page 23, that values are correct. Due to the edge $x \to ?$ in both *gen* set and *kill* set the set C remains unchanged while the edge from the *gen* set is added to the set E. This is as specified by the data-flow equations in figure 3.14 on page 41.

Following the control flow, the next three edges correspond to address-of assignments. The trace on the console shows that these assignments were identified correctly and that the *gen* sets, *kill* sets, and *strong* flags were computed according to the data-flow equations. This can also be seen in the data-flow values depicted in the CFG. The points-to edges from the location sets of the updated pointers to the location set `unk` are removed from the set C, and the corresponding edges to the location sets of the new targets are added. These new edges are also added to the set E.

The next control flow edge, from "after assignment 4" to "after assignment 5", represents the copy assignment `r = q`. Before that assignment, `r` pointed to an unknown location and `q` definitively pointed to `y`. The location set for `r` identifies a single location, and therefore the update is strong. Thus, after the assignment, the set C should contain an additional points-to edge $r \to y$, while the edge $r \to ?$ should no longer be present. Furthermore, the set E should contain the generated edge $r \to y$. As it can be seen in the annotated CFG, this is indeed the case, and hence the copy assignment is handled correctly.

The next statement is the load assignment `q = *a`. Since `a` points to `p` and `p` points to `x`, and before the assignment `q` pointed to `y`, the *kill* set must contain the points-to edge $q \to y$ and the *gen* set must contain the edge $q \to x$. The trace shows that the algorithm computed exactly these *gen* set and *kill* set and the CFG indicates that the sets C and E were updated in accordance with the data-flow equations.

The last statement is the store assignment `*a = r`. Since `a` points to `p` and `r` points to `y` the points-to edge $p \to y$ must be added to the sets C and E. Furthermore, the update is strong, and therefore the edge $p \to x$ must be removed from C. The annotated CFG shows that the implemented algorithm handled the store assignment correctly.

The analysis result for the basic assignments module shows that the implemented algorithm identified the module variables and created the corresponding location sets. Furthermore, the implementation handled the basic assignment statements according to the data-flow equations given in chapter 3. A comparison of the expected result of the analysis, depicted in figure 3.4 on page 19, and the actual result, depicted in figure 4.1, shows that the computed points-to graphs, C, match the expected points-to graphs. Therefore, this program confirms that the presented data-flow equations for basic assignment statements are sound and correctly implemented in the nesC compiler.

4.3 Control-Flow: Branching and Merging

In addition to basic assignment statements, the input language supports control-flow statements. A program that contains an `if` statement, and therefore has a branching point in the control-flow, was presented as an example in section 3.2.2. The program's source code is given in figure 3.5 on page 20, the computed points-to information is depicted in figure 4.2. Comparing the computed result of the analysis with the expected result in figure 3.6 shows that the correct points-to information was computed. The points-to information reaching the branching node, "after assignment 3", is propagated along both outgoing control-flow edges. At the join node, "if JOIN 5", the information from the branches is merged. This merge leads to a points-to graph that contains two points-to edges for `a`: $a \rightarrow p$ and $a \rightarrow q$. Therefore, the update `*a = p` can not be strong and this fact is correctly discovered by the implemented analysis.

Another construct of the input language that leads to branching and merging of the control-flow is `for` loops. A program that contains such a loop is given in figure 4.3, the analysis result is shown in figure 4.4.

This program reveals an imprecision in the analysis. The algorithm reports a possible assignment to an unknown location for the statement `*a = q`. Only in the first iteration of the loop, `a` is a null pointer. After the first iteration, `a` points to `p`. Since the assignment `*a = q` is guarded by a check that `a` points to a valid location, the program never writes to an unknown memory location. However, since the analysis is path-insensitive, it does not compute that information. Another imprecision, also due to path-insensitivity, is that the points-to edges $a \rightarrow ?$ and $p \rightarrow ?$ reach the exit node. Since the loop is executed two times, `a` definitively points to `p` and `p` definitively points to `x` at the exit node.

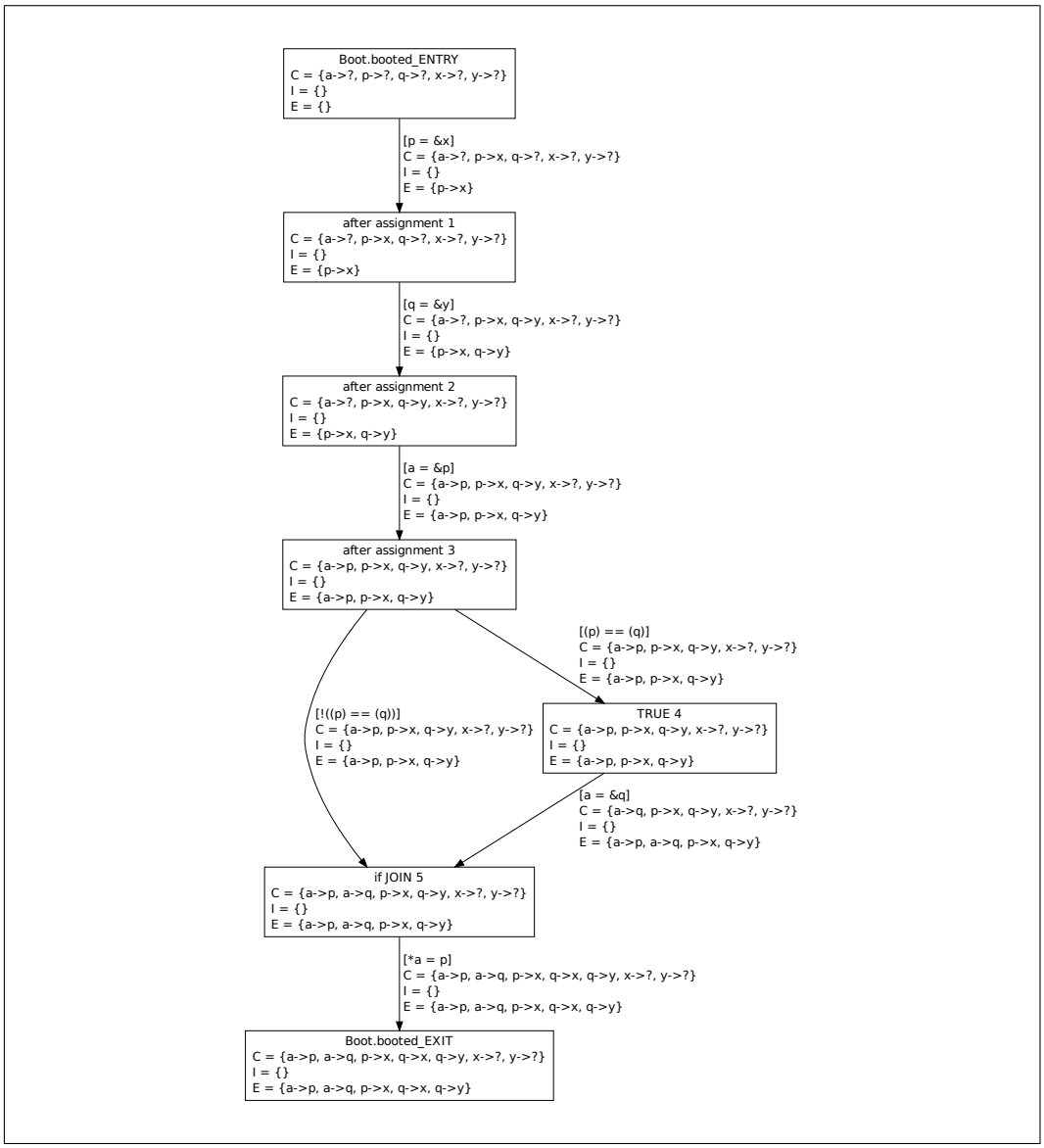

Figure 4.2: CFG for the program to evaluate the handling of branches in the control flow, annotated with parallel points-to information.

```
module ForC {
  uses interface Boot;
} implementation {
  uint8_t x;
  uint8_t *p;
  uint8_t *q;
  uint8_t **a;

  event void Boot.booted() {
    q = &x;

    for (x = 0; x < 2; x++) {
      if (a != NULL)
        *a = q;

      a = &p;
    }
  }
}
```

Figure 4.3: `ForC`: A module to test the analysis of loops.

4.4 Function Calls

A program that consists of three functions was presented in section 3.3.2. The source code is shown in figure 3.9 on page 28 and the expected result of the points-to analysis in figure 3.10. This program is used to evaluate the implementation of the interprocedural analysis. The result obtained by analyzing the program with the implemented algorithm is depicted for each function and context individually. Figure 4.5 shows the result for `Boot.booted`, figure 4.6 the result for `f` in the context of the call from `Boot.booted`, figure 4.7 the result for `g`, and figure 4.8 the result for `f` in the context of the call from `g`.

The obtained results correspond with the expected results, and that indicates that the implementation correctly handles function calls. An important observation is that the implemented analysis yields two different results for the function `f`, depending on the calling context. That demonstrates the context sensitivity of the analysis.

One feature of the presented analysis that is not implemented in the prototype is support for parameters and return values of functions. Therefore, the evaluation of the interprocedural analysis is limited to the effect of called functions on module variables and pointers, and hence this section does not present a program with functions that take parameters or return a value.

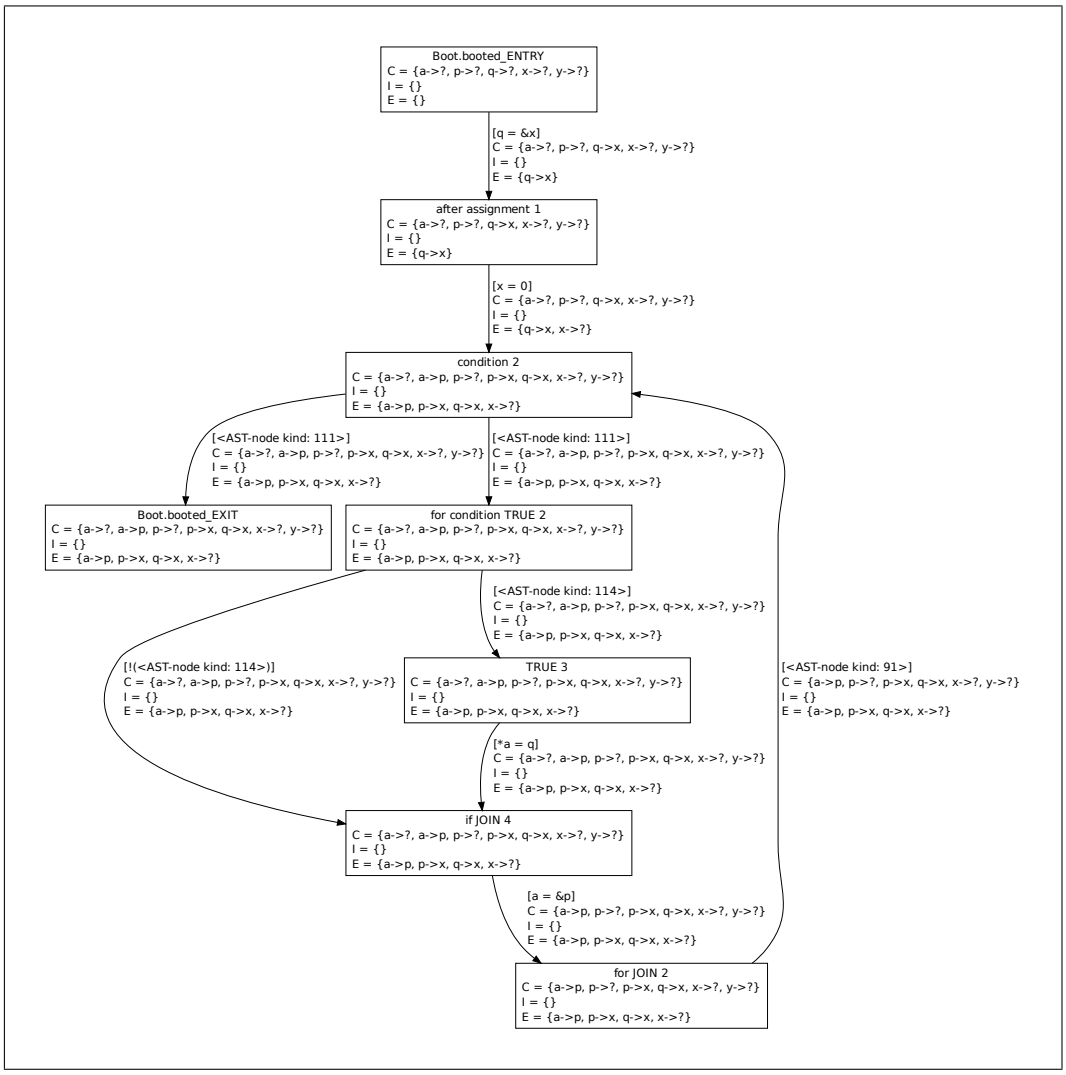

Figure 4.4: CFG for `Boot.booted` of `ForC`, annotated with parallel points-to information.

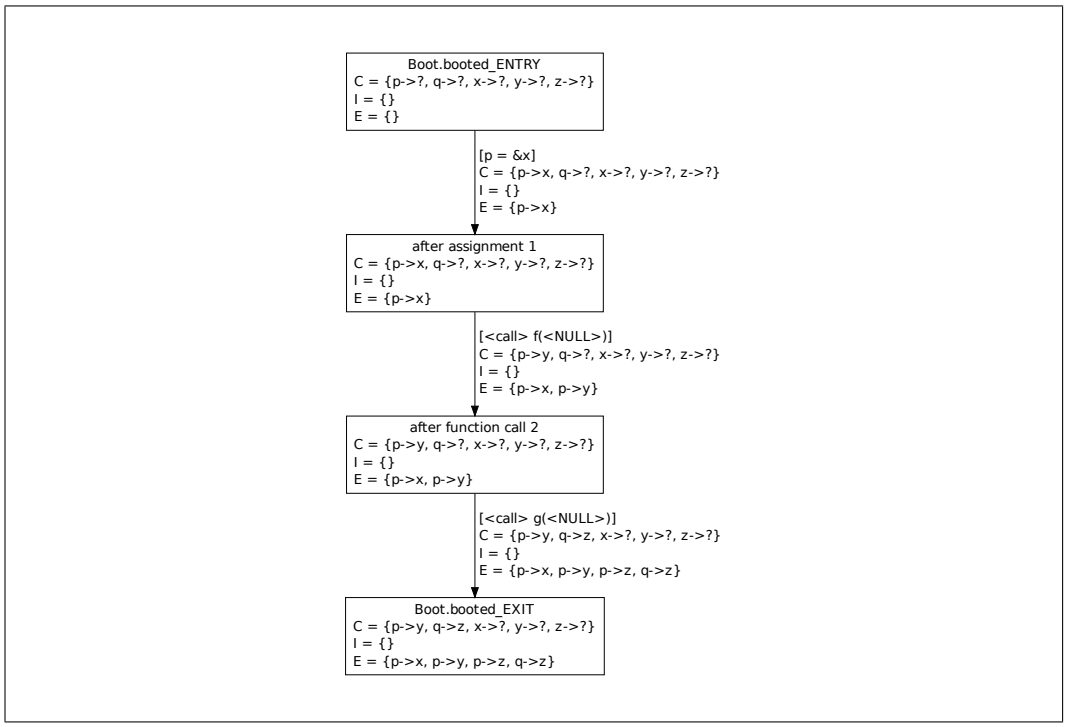

Figure 4.5: CFG for `Boot.booted` of the program to evaluate the interprocedural analysis, annotated with parallel points-to information.

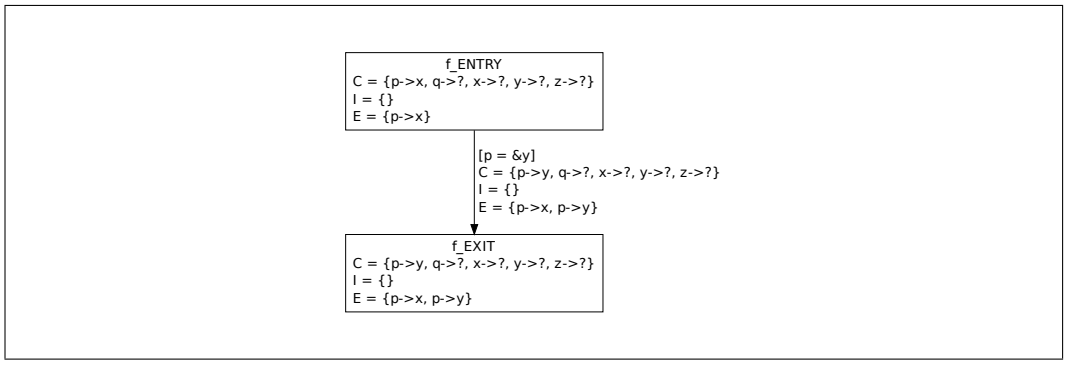

Figure 4.6: CFG for `f` of the program to evaluate the interprocedural analysis, annotated with parallel points-to information in the context of the call from `Boot.booted`.

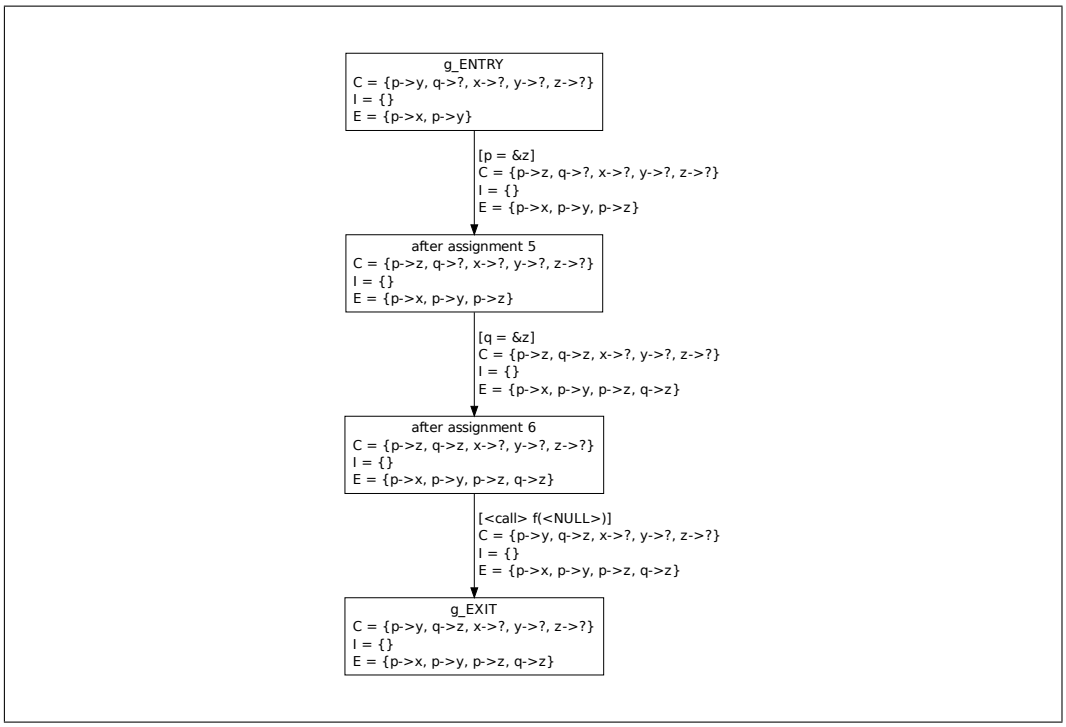

Figure 4.7: CFG for `g` of the program to evaluate the interprocedural analysis, annotated with parallel points-to information.

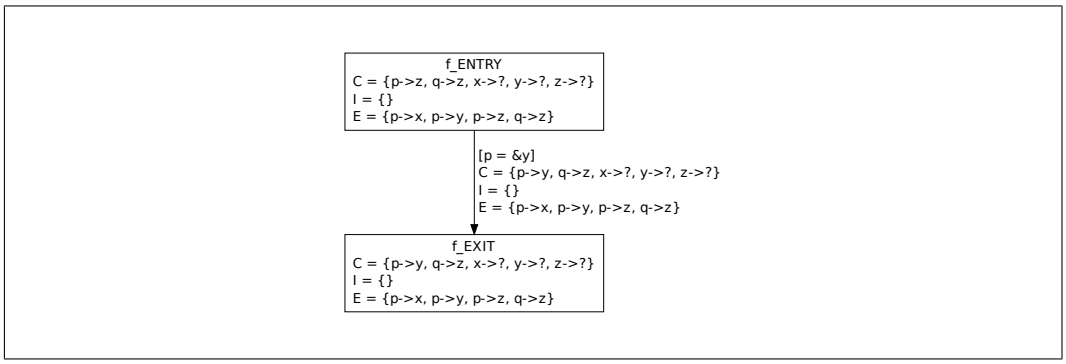

Figure 4.8: CFG for `f` of the program to evaluate the interprocedural analysis, annotated with parallel points-to information in the context of the call from `g`.

4.5 Events

The goal of the presented algorithm is the analysis of event-driven programs. To evaluate its capabilities, the implemented analysis is used to analyze the program presented in figure 3.12 on page 34. As in section 3.4.2, we will focus on the points-to relations on the event-level. The constructed event graph with the computed points-to information is depicted in figure 4.9. For comparison, the expected result is shown in figure 3.13 on page 35. Both graphs show the same result, they differ only in the positions of the parallel threads, and the computed event graph has additional identifiers on the "parbegin", "parend", "begin", and "end" nodes. Of special interest is the parallel construct that models the non-deterministic execution order of the events `RadioCtl.startDone`, `Radio.sendDone`, and `Timer.fired`. The depicted event graph shows that the interference information for the individual threads of events is computed properly and correctly merged into the current points-to graphs of the respective thread.

To evaluate the analysis of programs with recurring events, the event graph construction in the nesC compiler is instructed to treat `RadioCtl.startDone` and `Timer.fired` as recurring events. This results in the event graph depicted in figure 4.10. Treating `RadioCtl.startDone` and `Timer.fired` as recurring events leads to a "recbegin" and a "recend" node in the event graph that enclose the parallel threads of events. Since recurring events may execute in any order—one instance of a thread of events may even interfere with another instance of the same thread—any of the three concurrent events may be executed before any other concurrent event of the program. Therefore, the computed points-to graphs for the recurring events contain all points-to edges generated by all other recurring events. This is the desired result and indicates the correct implementation of the event-sensitive analysis.

4.6 Results

The evaluation showed that the implemented points-to analysis for event-driven programs correctly handles programs that adhere to the input language. An exception are programs that use structures or arrays, or contain functions that take parameters, return a value, or have local variables. Unfortunately, support for these features is currently missing from the implementation. The other constructs of the defined input language are handled correctly, and therefore we obtain the expected points-to relations for the example programs from chapter 3, as well as for some additional programs. It should be noted that the unsupported features are an implementation detail and not a limitation of the presented algorithm.

Another observation is that the analysis of example applications of the TinyOS distribution that contain pointers is infeasible with the current analysis. Contrary to our assumptions about analyzed programs, these applications pass pointers through

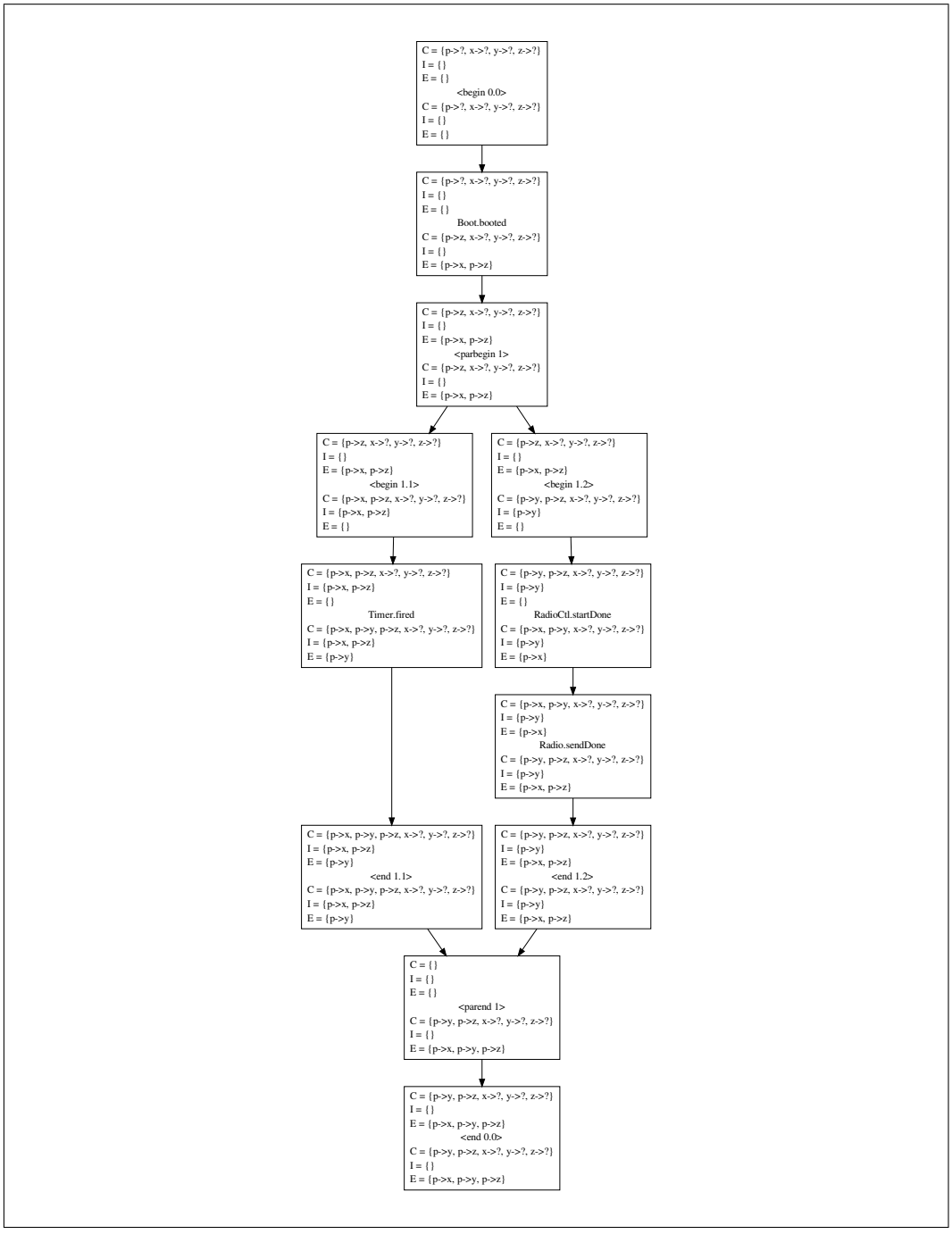

Figure 4.9: Event graph for the program to evaluate the event-sensitive analysis, annotated with parallel points-to information.

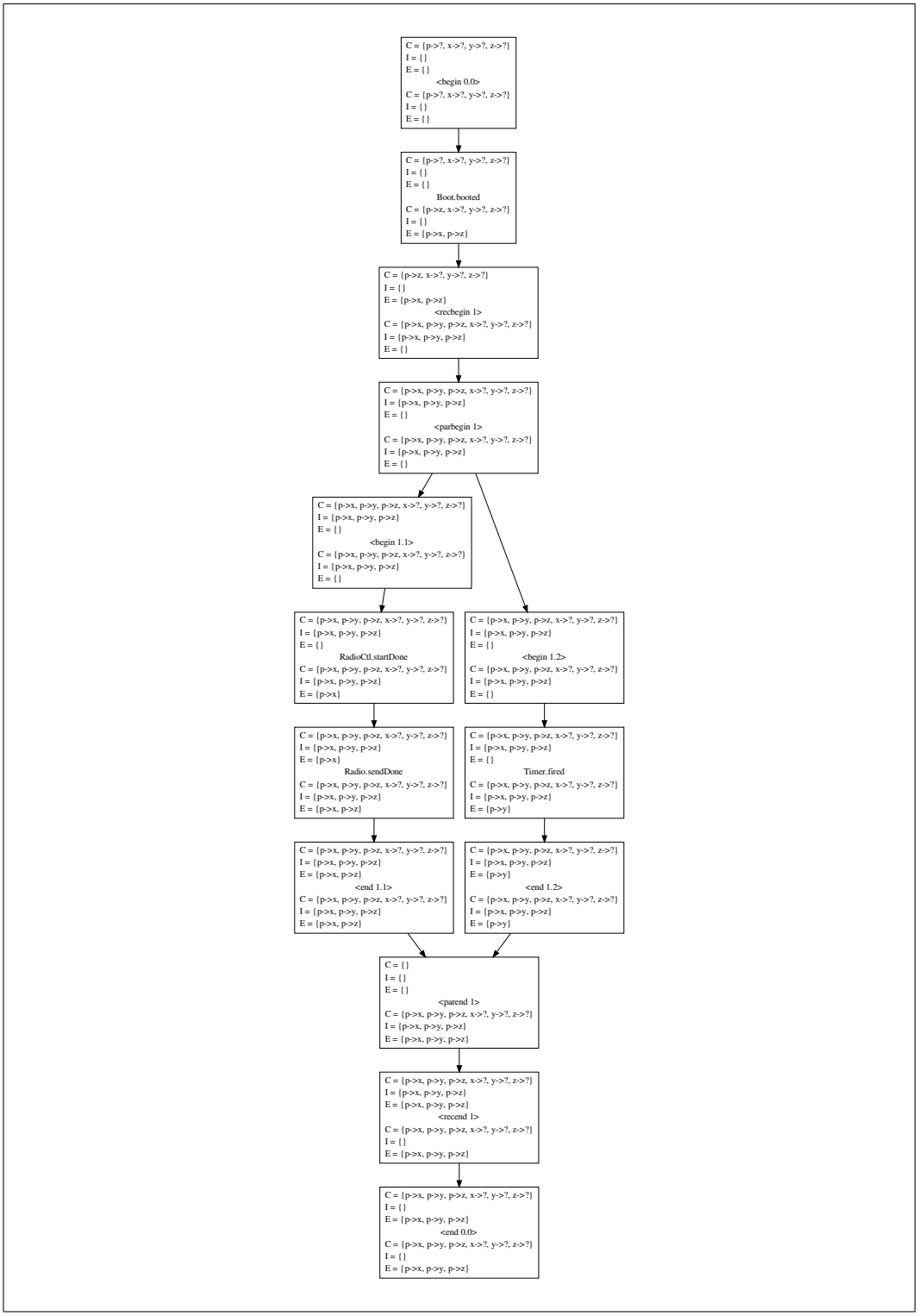

Figure 4.10: Event graph for the program with recurring events.

module boundaries. Since the analysis considers the top-level module of an application only, it can not discover modifications to the points-to relations outside of the analyzed module. Furthermore, typecasts are rather common in TinyOS applications, but neither does the defined input language contain typecasts nor are statements that contain typecasts handled by the implementation. These limitations will be discussed in the next chapter, together with ideas to improve the situation.

5 Limitations, Conclusions, and Future Work

The evaluation showed that there are some limitations. Some arise from the limited input language, some from limitations of the implementation, some from limitations of the available infrastructure in the nesC compiler, and some from our initial assumptions about the analyzed programs. In this chapter, first, these limitations are discussed and ideas to improve the situation are presented. Then, some ideas for future research are provided, and finally, this chapter concludes with a summary of the work and the obtained results.

The input language for the intraprocedural analysis allows only `if`, `for`, and basic assignment statements. At first glance, this is not a real limitation since other forms of assignment and control-flow statements can be transformed into statements allowed by the input language. However, an important feature that is missing from the input language is typecasts. Many existing programs use typecasts and therefore this is a real limitation. To enable the analysis of such programs, typecasts should be supported by the input language. Fortunately, the analysis uses location sets to represent blocks of memory and is independent of a pointer's or variable's type. Therefore, support for typecasts appears to be a straightforward extension to the input language and the implementation in the compiler (that must handle the corresponding AST nodes), while the analysis algorithm itself remains unchanged.

The current implementation does not consider parameters, return values, and local variables of functions. These were left out of the implementation due to time constraints on this thesis. The presented analysis algorithm supports parameters, return values, and local variables, by mapping the points-to graph at the call site into the scope of the called function, as described in section 3.2.4. Therefore, support for parameters, return values, and local variables seems to be merely an implementation detail.

Another limitation of the implementation is the missing support for structures and arrays. To compute the location sets for structures and arrays, the offset and stride values must be known. Unfortunately, these values are unavailable in the nesC compiler that transforms nesC code into C code and does not cover the layout of objects in memory. Whether the required values can be obtained within the implementation in the compiler or must be provided externally remains to be investigated.

The restriction of the points-to analysis to the top-level module of a program turned out to be a major limitation in the analysis of TinyOS applications. A manual inspection of the example applications in the TinyOS distribution showed that the pointers used in top-level modules reach that module as parameters of events or as return values of command calls. Furthermore, the top-level modules pass the pointers as parameters to commands. Therefore, the assumption that code in other modules has no effect on the points-to relations in the analyzed module does not hold for these applications. To analyze such applications, the points-to relations must be tracked across module boundaries. This is no problem for the presented points-to analysis, but the current implementation relies on the megagraph provided by the NGE extension of the nesC compiler. Since this compiler extension provides the megagraph for the top-level module only, some work is required to extend the points-to analysis across module boundaries.

The presented analysis is not path-sensitive, i.e., it does not consider branching conditions. Path-insensitivity is a common property of data-flow analyses, and, in general, this is not a major limitation. Whereas making the analysis completely path-sensitive requires techniques beyond data-flow analyses, there is one case that can be handled with the available points-to information and may improve the precision of the result: considering branching conditions like "`if (p != NULL)`." The current analysis displays warnings about accesses to unknown memory locations due to uninitialized pointers, even if the pointer dereference is guarded by a check that it is properly initialized. Thus, investigating the inclusion of such simple pointer equality and inequality checks may be worthwhile.

One minor point is that the analysis does not handle concurrent events as completely atomic. If a pointer `p` is first assigned the address of a variable `x` and then later in the same event is assigned the address of a variable `y`, both points-to edges $p \to x$ and $p \to y$ are included in the set of generated edges. Therefore, both edges are in the interference graph of the parallel threads of events, even though the points-to edge $p \to x$ is never visible outside the event, due to its atomic execution. Events can be handled completely atomic by introducing separate data-flow equations for event nodes. The analysis of an event would always start with an empty set E. The points-to edges from the *gen* set would be added to the set E as before, but additionally, the edges from the *kill* set would also be removed from the set E by a strong update. Finally, on the event-graph level, the set E' after the event node would be merged with the set E before the event. This results in completely atomic events. However, the presented analysis is sound and handling events completely atomic may only increase the precision. Therefore, the precise handling of atomic events was deferred until it is investigated in the broader context of asynchronous events and traditional threads.

In addition to synchronous, atomic events, nesC also supports asynchronous events that may interrupt the execution of other events. Hence, future work should extend the presented algorithm to support asynchronous events. The current imprecise

handling of atomic events may be a good starting point in that direction. Furthermore, there is support for threads in TinyOS [16]. Since the presented analysis for event-driven programs is based upon an analysis for multithreaded programs, the integration of threads into the analysis seems possible. Therefore, an extensive analysis that covers synchronous and asynchronous events as well as threads should be explored in future research.

In conclusion, this thesis presented an interprocedural, flow-, and context-sensitive points-to analysis for event-driven programs. The novel feature of this analysis is the event graph to model the dependencies of events. Sequences of concurrently executing events are modeled as parallel threads of events. This enables the analysis of event-driven programs with an algorithm based upon Rugina and Rinard's points-to analysis for multithreaded programs. For every thread of events, the algorithm computes the interference information from parallel threads and merges it with the points-to information computed for the current thread. Additionally, this thesis argued that the presented analysis is sound and terminates. The evaluation showed that the algorithm and its implementation in the nesC compiler correctly analyze programs written in the presented input language. There are some limitations, but most of them are due to missing features in the implementation. Future support for typecasts, parameters, and return values, as well as a whole-program analysis that lifts the restriction to the top-level module of an application will improve the applicability to real world TinyOS applications. The presented analysis is not limited to the nesC programming language. It is also a valuable instrument for the analysis of event-driven programs written in other programming languages. One example is applications with a graphical user interface that react to events triggered by user input.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, 2nd Edition*. Pearson/Addison Wesley, 2007.

[2] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2-3):79–111, 1999.

[3] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[4] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 115–125, New York, NY, USA, 2003. ACM.

[5] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.

[6] David Gay, Philip Levis, David Culler, and Eric Brewer. *nesC 1.3 Language Reference Manual*, 2009.

[7] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM.

[8] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 1–15, New York, NY, USA, 1996. ACM.

[9] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *ACM SIGPLAN Notices*, 35(11):93–104, November 2000.

[10] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis*

for Software Tools and Engineering, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[11] Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 339–350, New York, NY, USA, 2007. ACM.

[12] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[13] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[14] Uday P. Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.

[15] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[16] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Răzvan Musăloiu-E, Philip Levis, Andreas Terzis, and Ramesh Govindan. TOSThreads: Thread-safe and non-invasive preemption in TinyOS. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 127–140, New York, NY, USA, 2009. ACM.

[17] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[18] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the 18th annual ACM Symposium on Principles of Programming Languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM.

[19] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM.

[20] Philip Levis and David Gay. *TinyOS Programming*. Cambridge University Press, 2009.

[21] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[22] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 77–90, New York, NY, USA, 1999. ACM.

[23] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.

[24] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis*, chapter 7, pages 189–233. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[25] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[26] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[27] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[28] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 1–12, New York, NY, USA, 1995. ACM.