Bachelor's Thesis Computing Science

Atomic Tasks in Task-Oriented Programming

Niek Adams

July 2025

First supervisor/assessor:
dr. Mart Lubbers

Second assessor:
dr. Peter Achten

Radboud University

Abstract

Energy harvesting uses sustainable energy sources to replace batteries in computers that are part of the Internet of Things. These energy sources are usually unreliable, so intermittent computing is introduced to maintain progress when working with these inconsistent sources of power. However, some tasks should restart after losing power without making progress or reverting to an intermediate state. We call these tasks atomic tasks. This thesis answers the question of how atomic tasks can be implemented in mTask, a Task-Oriented Programming framework. The implementation of intermittent computing in mTask is analysed to design atomic tasks that fit in the existing framework. Our main contributions are a semantics for atomic tasks and an implementation in mTask, which we evaluate using a case study.

Contents

Chapter 1

Introduction

A growing number of Internet of Things (IoT) devices, issues with grid congestion and a rising demand for batteries ask for a more sustainable approach towards the development of IoT systems. Research is done towards sustainable solutions, but these solutions are not always adequate for real-world use cases. Energy harvesting is a sustainable alternative to the use of batteries by drawing energy from the environment, such as solar power or kinetic energy. Our goal is to expand the set of problems that can be solved using energy harvesting.

1.1 Task-Oriented Programming

We focus on Task-Oriented Programming (TOP), a programming paradigm that revolves around the idea of executing tasks [15]. This paradigm sees computer programs as collections of tasks that are to be performed. We look at mTask: a TOP framework specifically designed for developing IoT systems [6]. Take a smart sunshade that is connected to a server to receive input on whether to open or close. This sunshade can be represented by a sequence of tasks, listed in Listing 1.1, that is executed repeatedly. The advantage of TOP is that developers only state the sequence of tasks to be executed, and the server automatically generates an interactive web application and connections with edge devices. Additionally, mTask generates the code to run on edge devices, based on the task composition. TOP and mTask are further explained in Chapter 2.

```
1. Wait for server input
2. Start the motor
3. Wait for the shade to be fully closed or opened
4. Stop the motor
```

Listing 1.1: Task composition of a smart sunshade.

1.2 Intermittent computing

The sunshade from Listing 1.1 could be attached to the energy grid, or it could contain a battery. However, this is not as sustainable as using energy harvesting. By implementing a solar panel, it is possible to use renewable energy, thus not exhausting our planet's resources. But, solar power is unreliable: power could be lost at any moment. The mTask framework supports intermittent computing, allowing the system to restore to its previous state after losing power [17]. This restores the established network connection with the server and the tasks that are to be executed. For example, losing power during Task 2 of the smart sunshade restores the state such that the sunshade continues with starting the motor and reusing the established connection with the server. So, intermittent computing allows developers to implement sustainable IoT devices that use replaceable energy.

However, there is an issue with the current implementation of intermittent computing that limits its applicability to real-world use cases. Take the smart sunshade example. If power is lost during Task 3, the system restores to the state where it is waiting for the shade to open or close, without starting the motor again. In other words, we have to wait forever, because the shade will not fully open or close. Another example where the current implementation of intermittent computing is not yet sufficient is a noise sensor in the woods. A sensor measures the noise level every 5 minutes and warns a ranger after 3 consecutive measurements where the noise exceeds a limit. These measurements are time-sensitive information because the information is only useful for a limited amount of time. There is no point in warning the ranger one hour after the incident, so measuring should restart after losing power without restoring intermediate measurements.

1.3 Sustainable IoT with atomic tasks

The general issue with the current implementation of intermittent computing in mTask is that certain sequences of tasks should not make progress when losing power. This could be the case with time-sensitive information or responding to user input. This limits the use cases where intermittent computing can be used, and consequently, where energy harvesting can be used as a sustainable alternative to batteries and grid power.

To solve this issue, we suggest extending mTask with atomic tasks. The purpose of atomic tasks is to combine tasks into an atomic task that should not make progress when losing power. Instead, it should restore as if the atomic task was never started. The task is called atomic because it should be indivisible by intermediate checkpoints. Take the example of the smart sunshades from Listing 1.1, where Task 2 and Task 3 are now combined in an atomic task. This is demonstrated in Figure 1.1.

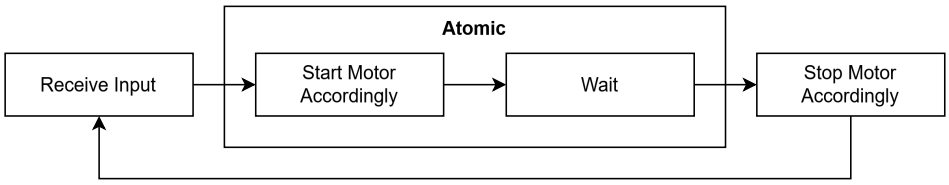

Figure 1.1: A task composition of smart sunshades using atomic tasks.

1.4 Research contributions

This thesis solves the general issue that the current implementation of intermittent computing cannot be used for cases that involve time-sensitive information and input interaction. A solution to this issue is not yet available for TOP. The research question is: How can we implement atomic tasks in mTask? Answering this question results in a larger set of problems that can be solved sustainably and reliably, because atomic tasks give developers more control over the behaviour after restarting from a power loss. This thesis is relevant for developers of IoT systems and researchers in the fields of sustainable innovation, IoT and TOP.

The contributions of this thesis are listed below. Chapter 4 additionally compares the topics covered in this thesis to related concepts in computing science to provide context for our solution.

- We analyse and report on the current implementation of intermittent computing in mTask (Section 3.1).
- We implement atomic tasks as a type in the mTask server to allow developers to define sequences of tasks that have to be executed atomically (Section 3.2).
- We propose a semantics for atomic tasks and modify the mTask client such that atomic tasks are restored to their initial state, in order to revert progress (Section 3.3).
- We evaluate the implementation of atomic tasks in mTask and show that they solve the introduced problem (Section 3.4).

Chapter 2

Task-Oriented Programming

TOP is a programming paradigm where the developer defines programs in terms of tasks. A task represents work to be completed, either by a human or a machine, and compositions of tasks create meaningful applications [10]. The main advantage of TOP is that the entire application is generated from a single source. In other words, the entire application, including the user interface, application logic and network connections, is generated from only one source where the developer defines sequential or parallel task compositions. So, the developer is only concerned with one type system and one compiler for the entire application, making it more intuitive to develop complex systems.

One framework implementing TOP is iTask. It offers a high level of abstraction and generates web applications based on task compositions and their types [14]. The iTask framework is implemented in the functional programming language Clean as a Domain-Specific Language (DSL). A DSL is a language created for a specific purpose, allowing for clearer instructions and limiting unnecessary overhead [5]. Clean is similar to Haskell with slight variations in syntax and semantics [9].

2.1 Tasks and task values

To explain the purpose of tasks in TOP, we will use a cooling system as an example. The cooling system measures the temperature and turns a cooler on if and only if the temperature exceeds 22 degrees Celsius. Pseudo code for a function `coolingSystem` that does this is given in Listing 2.1. Line 2 shows a task that turns the cooler on or off, depending on the value of `state`. The task `readTemperature` in Line 3 reads from the Digital Humidity and Temperature (DHT) sensor, and this value can be used as input in Line 4.

```
1 coolingSystem = \state → ( // state is boolean input
2         setCooler state // turns the cooler on if state is true
3     ⇒   readTemperature // Read from the DHT sensor
4     ⇒   \temperature →
5             if temperature > 22:
6                 coolingSystem True // Recursive call to coolingSystem
7             if temperature <= 22:
8                 coolingSystem False
9     )
```

Listing 2.1: Pseudo code for a cooling system.

In TOP, each task has input and observable output and may cause side effects whilst executing. The output of a task is a task value, and these task values have stability: No Value, Unstable or Stable. A task with no value is not yet complete. An unstable task has a value, but its value could change in the future. For example, reading from a DHT sensor, as done in Line 3, always results in an unstable task value, because the temperature can be different when observing the task at a later point in time. Finally, when a task is stable, it has finished execution, and its value should never change. Observing a task value means that a task regularly reads the task value of another task and acts on this value, based on a condition. For example, the task in Line 4 of the pseudo code observes the value from the task in Line 3 to see if the temperature exceeds 22 degrees or not. Only when a condition is met, the program continues. In this example, a condition will always be met, but this property can also be used to wait for user input. In conclusion, tasks have a task value that can be observed by other tasks to influence the behaviour of the application.

2.2 Connecting TOP and IoT using mTask

It is possible to use TOP for developing IoT systems by using mTask: a framework that extends iTask with support for embedded devices [6]. The mTask framework contains a DSL integrated in Clean together with a runtime system for microcontrollers. Like iTask, mTask also offers the benefit of a single source where developers create the application. The application then generates a web interface through iTask. The mTask framework consists of a byte code generator that generates client instructions and a client that interprets these instructions on microcontrollers. So, mTask builds on iTask's high level of abstraction to allow creating scalable IoT applications from a single source.

Tasks in iTask can interact with Shared Data Sources (SDSs), which can be a database, shared file, or even a utility like an API. Given the embedded nature of the IoT domain, mTask also provides support for the use of peripherals which behave similarly to SDSs and are implemented as tasks. These peripherals include temperature sensors or digital pins that the client can either read from or write to. Figure 2.1 shows a diagram of the interaction between the mTask server and the mTask client. The server and the client exchange byte code instructions and results, and the two may influence each other's behaviour through SDSs.

Listing 2.2 and Listing 2.3 display the code for a working mTask application for the cooling system from Listing 2.1. The server and the client of mTask are explained using this code.

- Listing 2.2 shows the tasks that are processed on the server. The mTask server is explained in Section 2.3. Line 9 contains the type definition of `coolingSystemTask`, a task to be executed on a client.
- Listing 2.3 shows the function `coolingSystemTask` that is executed on a client. Section 2.4 explains how the mTask client interprets these instructions and executes the task.

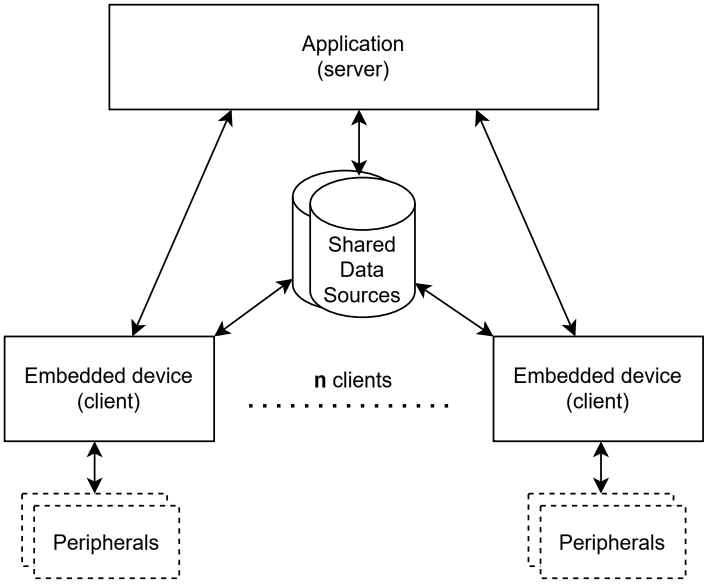

Figure 2.1: Interaction between the mTask server and clients.

2.3 Defining tasks in the mTask server

The mTask server is a Clean library that is used in combination with iTask to let developers create complete IoT applications using the TOP paradigm. The mTask server is a compiled web application that runs on a server or desktop environment. Running the code from Listing 2.2 results in a web application that allows connecting with mTask clients using TCP. The currently supported connection protocols in mTask are TCP, MQTT, and serial. Line 4 shows a task that is to be completed by a human: `enterInformation`. This task lets the user enter the TCP connection details for the mTask client. After connecting, a task is *lifted* to mTask. Lifting a task involves generating byte code instructions for the client and sending these to the device. The client then interprets these instructions and runs the application. Line 6 shows how the instructions for the cooling system are lifted to the client `dev` using the function `liftmTask`. The instructions for the client are given in `coolingSystemTask`, which is explained in Section 2.4. Finally, the web page shows a button to terminate the application, as stated in Line 7.

```
1  Start w = doTasks main w
2
3  main :: Task ()
4  main = enterInformation [] <<@ Title "Device details"
5      >>? \dd=:{TCPSettings|host,port}→withDevice dd \dev→
6          liftmTask coolingSystemTask dev
7      >>* [OnAction (Action "Stop") (always (return ()))]
8  where
9      coolingSystemTask :: Main (MTask v ()) | mtask, dht v
```

Listing 2.2: Server code for an automated cooling system in mTask.

```
9  coolingSystemTask :: Main (MTask v ()) | mtask, dht v
10 coolingSystemTask = dht (DHT_DHT (DigitalPin D4) DHT11) \dht =
11     declarePin D6 PMOutput \coolingPin =
12     fun \checkTemperature = (\st→
13               writeD coolingPin st
14         >>|. temperature dht
15         >>*. [IfValue (\m → m >. (lit 22.0) &. Not st)
16                       \_→checkTemperature true,
17               IfValue (\m → m <=. (lit 22.0) &. st)
18                       \_→checkTemperature false]
19     ) In {main=checkTemperature false}
```

Listing 2.3: Client instructions defined in an mTask server.

2.4 Running tasks in the mTask client

The server generates byte code instructions for the task `coolingSystemTask` in Listing 2.3 and sends these instructions to the mTask client. The mTask client is a compiled C program that runs on microcontrollers as a runtime system [8]. The most important jobs of the runtime system are managing the device's memory and executing the received tasks.

The tasks in Listing 2.3 are converted to byte code. This code contains basic tasks, like `temperature dht` in Line 14. Basic tasks are tasks that cannot be expanded any further, and they represent work that needs to be done. Next to basic tasks, there are task compositions. Combinators such as `>>*.` and `>>|.` are used to combine multiple tasks into a sequential composition. Parallel combinators like `.&&.` and `.||.` exist to perform multiple tasks in parallel and either combine the results or keep only one result. The server sends the client instructions for these tasks and combinators.

The first step after receiving a new task in the client is interpreting the received instructions. The interpreter is responsible for doing this. In the cooling system, it would start with evaluating `checkTemperature false` from Line 19. By evaluating the expression, a task tree is created that can later be used to execute tasks. An example of a task is `writeD coolingPin st` in Line 13 that turns digital pin `D6` on or off, depending on the boolean evaluation of `st`. The interpreter strictly evaluates everything that can be evaluated, and it ignores the right side of a sequential combinator, because there could be multiple continuations, as is visible in Lines 15–18, where the outcome is undecided and can be `checkTemperature true` or `checkTemperature false`. In conclusion, the interpreter does not execute the tasks itself, but evaluates expressions and places them in a task tree.

Task trees are the runtime representation of tasks, where each task has zero or more child nodes. Take the parallel combinator `.&&.`, which has two children: the left and the right side of the combinator. Figure 2.2 shows

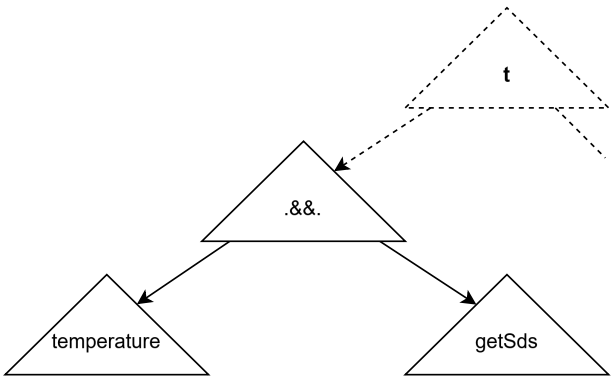

Figure 2.2: Task trees in the mTask client.

an example task tree representation of an expression containing this parallel combinator. This figure shows an arbitrary task combinator **t** that contains `.&&.` as one of its child nodes. This parallel combinator combines the results of its child nodes in a tuple. In this example, it reads the temperature from a sensor and a value from an SDS. This is useful in a thermostat where **t** observes both the actual temperature and the desired temperature, which may change. The purpose of these task trees is to process tasks from the root of the tree. Apart from child nodes, task trees contain supporting information, such as a pointer to the parent tree, scheduling information, and specific information required to perform basic tasks. Executing a task is done by the rewriter.

The rewriter executes tasks by traversing the task tree and its child nodes. The rewriting is done in small steps until the task has a stable task value. Before rewriting, the client pops from the scheduling queue the next task to be executed and starts the rewriter in the root of this task's task tree. During the rewriting, it may be necessary to call the interpreter. For example, when the right side of a sequential combinator can be evaluated. Because the rewriting is done in small steps, different tasks are seemingly performed in parallel by interleaving the rewriting steps of different tasks.

Apart from interpreting instructions and rewriting tasks, the client also manages the memory by storing the stack for interpretation and rewriting, the task trees, and auxiliary task details, such as peripheral information. To minimise memory usage, task trees can be marked as trash, and a garbage collector regularly cleans the heap. In conclusion, the mTask client is concerned with receiving tasks from the server and managing the execution on the microcontrollers.

Chapter 3

Implementing Atomic Tasks

The goal of an atomic task is to revert any progress made in the child task when the IoT device loses power. There is a difference between the terms *atomic task* and *atomic combinator*. In mTask, combinators create compound tasks by combining tasks [8]. The atomic combinator allows exactly one child task, which can also be a composition. We refer to the combination of the atomic combinator with its child task as an atomic task. Figure 3.1 shows the difference between these terms in a piece of code where the atomic task encapsulates another task. The atomic combinator describes that the child task should be reset to its original state when power is lost during execution.

In this chapter, we explain how the atomic combinator can be implemented in the mTask system. The first step in implementing the atomic combinator is understanding the current implementation of intermittent computing in mTask (Section 3.1). The client depends on the server for generating the instructions, so the next step is implementing the instruction `atomic` in the mTask server (Section 3.2), after which we describe the intended behaviour and implement this in the client (Section 3.3). Finally, we evaluate this implementation (Section 3.4) using the case study with smart sunshades from the introduction, to verify that the initial problem with time-sensitive information and user interaction is solved.

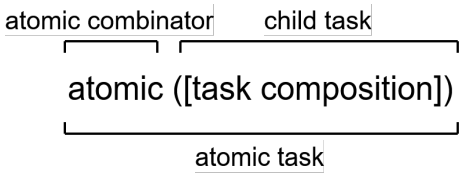

Figure 3.1: Terminology of atomic tasks.

3.1 Analysing intermittent computing

Intermittent computing is implemented in mTask by creating checkpoints after each rewrite step [17]. Checkpointing allows the client to restore its state after losing power, hence preserving any progress. The checkpoint encodes the state of the client in a way that the client can restore this state after rebooting. Checkpointing allows the client to progress even when power is lost during the execution of the program. This makes checkpointing a suitable method for implementing intermittent computing.

Intermittent computing in mTask is implemented implicitly, meaning that checkpoints are made automatically after rewriting a task tree. The client stores a checkpoint of its state in persistent memory, such as EEPROM or FRAM, or in a file when running in a desktop environment. The following information is captured in a checkpoint: tasks, task trees, the scheduling queue, and memory pointers for the task, heap and events. Additionally, the memory address of the data is stored, in case the pointers are adjusted for Address Space Layout Randomisation (ASLR). ASLR is a security measure that randomises memory addresses, and this should be accounted for when restoring from power loss. Restoring from power loss is similar to storing the checkpoint, in reverse order. The stored information is placed in Random Access Memory (RAM) again, and the pointers are adjusted for ASLR if necessary. To prevent invalid checkpoints when power is lost whilst writing a checkpoint, the current implementation uses double buffering, so the old checkpoint stays valid until a new checkpoint is written. Checkpoints store this validity as well.

3.2 Implementing atomic tasks in the server

The first step in implementing atomic tasks in the mTask framework is to implement them in the mTask server. Developers of IoT systems use the server library to create mTask applications, as mentioned in Section 2.3. The developer states the intended behaviour for the mTask clients in the server, and the server translates the instructions to byte code instructions for the client to process and execute. So, we extend this library with an atomic combinator to create atomic tasks. The mTask server is a Clean library, and implementing the atomic combinator is accordingly done in Clean. This section explains the changes to the server library that allow developers to create atomic tasks using the atomic combinator.

Currently, there are three types of combinators in mTask: sequential, parallel, and repeat. Sequential task combinations pass the result of the first task to the second task, parallel task combinations process two tasks simultaneously, and the repeat combinator repeats tasks indefinitely. The atomic and repeat combinators are both unary and they have the same type.

3.2.1 Class definition

To allow developers to make use of the atomic combinator, the class `atomic` is defined as shown in Listing 3.1. In this definition, `MTask v a` represents a task where `a` is the type of the task, such as an integer or boolean, and `v` is an interpretation. The mTask frameworks support three different interpretations: `Show`, `TraceTask`, and `Interpret`. The latter is most important to us, as it generates byte code instructions that the client can use. To support generating these byte code instructions for the atomic combinator, we declare an instance for `atomic` with `Interpret` as shown in Listing 3.2. The auxiliary function `tell` is used to append the byte code instructions, and `BCMkTask` instructs to create a new task node for the atomic combinator. Finally, the argument `t` represents the inner task and precedes the `atomic` instruction in the byte code. This is due to the reconstruction of the byte code instructions, where the client first generates the inner task tree before generating the atomic task tree. Section 3.3 explains the benefit of this approach. These additions to the server library allow developers to use the atomic combinator in task compositions.

```
class atomic v where
    atomic :: (MTask v a) → MTask v a
```

Listing 3.1: Class definition of the atomic combinator.

```
instance atomic Interpret
where
    atomic t = t >>| tell [BCMkTask BCAtomic]
```

Listing 3.2: Instance declaration for `Interpret` in `atomic`.

3.3 Implementing atomic tasks in the client

The client code runs on edge devices and receives byte code instructions published by the server. The runtime system is responsible for creating the associated task trees and rewriting them to update the task value. The actual semantics of the atomic task combinator should therefore be implemented in the client's runtime system. This section explains the intuition, semantics and implementation of the atomic task combinator in the client.

3.3.1 Intuition

The intuition behind the implementation of the atomic combinator is a task tree with two child nodes. One node contains the initial and untouched inner task, and the other node is the active child task. Rewriting an atomic task is done in the active child task until the atomic node is removed after using its unstable value, for example, or in a parallel composition. When the

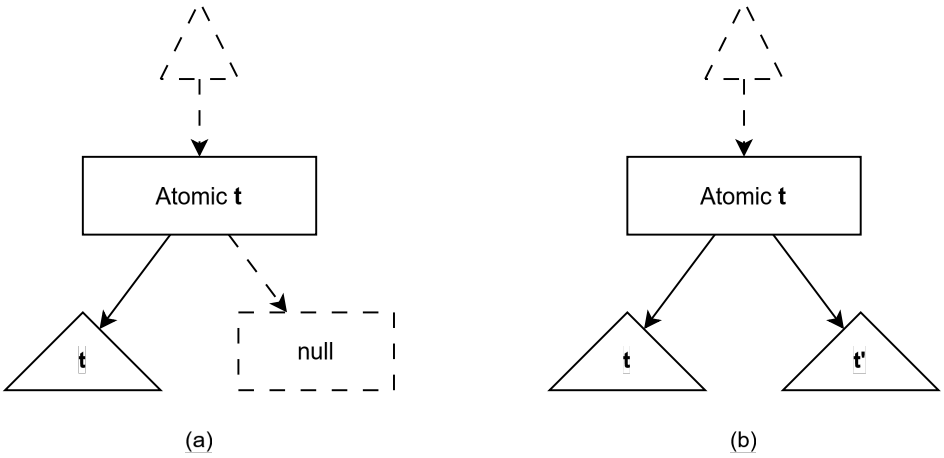

Figure 3.2: Intuitive view of an atomic task tree for all task trees **t**.

atomic task is rewritten after losing power, the active child task is replaced with the initial child task to reset any progress made in the active child task. The structure of the atomic task tree is visible in Figure 3.2, where (a) is the initial task tree after making the task node, and (b) is the task tree when rewriting the atomic task after duplicating $\mathbf{t}$ to $\mathbf{t}'$. Rewriting happens in $\mathbf{t}'$, and the value from $\mathbf{t}'$ can be observed by the parent of the atomic node.

3.3.2 Semantics

The goal of atomic tasks is to revert progress made within the child task after losing power. During normal operation, the active child task is rewritten, and its task value can be observed. However, if power is lost, the runtime system should replace the active child task with the initial child task, hence nullifying any progress and restarting the atomic task. This behaviour is similar to the repeat combinator, as both `atomic` and `repeat` are unary combinators capable of resetting their child task to its initial expression.

Big-step semantics for the atomic combinator are given in Table 3.1. The semantics are based on the notation from TopHat: a fully formalised language that implements the principles of TOP [16]. Different from TopHat, mTask accepts input before completely rewriting a task [1]. This means that we rewrite a task (t) to another task (t') instead of to a normalised task (n). We also omit dirty references (δ), because mTask does not keep track of these after rewriting a task. The semantics for the atomic combinator include the SDSs and peripheral input in the state (σ). The following definitions are used for the semantics.

- $t :=$ Task
- $t \downarrow t' :=$ Rewriting t results in t'
- $a(t_0, t_1) :=$ Atomic task with initial child t_0 and active child t_1
- $s :=$ Task stability $\in$ {stable, unstable, no value}
- $S := t \rightarrow s$ (function to retrieve task stability from a task)
- $\sigma :=$ State (including SDSs and peripheral input)
- $p :=$ Power state $\in \{\circ, \bullet\}$, where $\circ$ indicates normal operation and $\bullet$ is set after restoring from power loss.
- $P := (\sigma,\ a) \rightarrow p$ (retrieve the power state from an atomic task)

$P(\sigma,\ a(t_0,\ t_1)) = \circ$	
$\dfrac{t_1,\ \sigma\ \downarrow\ t_1',\ \sigma'}{a(t_0,\ t_1),\ \sigma\ \downarrow\ t_1',\ \sigma'}$	$S(t_1') = \text{stable}$
$\dfrac{t_1,\ \sigma\ \downarrow\ t_1',\ \sigma'}{a(t_0,\ t_1),\ \sigma\ \downarrow\ a(t_0,\ t_1'),\ \sigma'}$	$S(t_1') \neq \text{stable}$
$P(\sigma,\ a(t_0,\ t_1)) = \bullet$	
$\dfrac{t_0,\ \sigma\ \downarrow\ t_0',\ \sigma'}{a(t_0,\ t_1),\ \sigma\ \downarrow\ t_0',\ \sigma'}$	$S(t_0') = \text{stable}$
$\dfrac{t_0,\ \sigma\ \downarrow\ t_0',\ \sigma'}{a(t_0,\ t_1),\ \sigma\ \downarrow\ a(t_0,\ t_0'),\ \sigma'}$	$S(t_0') \neq \text{stable}$

Table 3.1: Proposed semantics for rewriting the atomic combinator.

3.3.3 Modifications

The client is written in C and depends on the mTask server library for generating all possible byte code instructions in the enum `BCInstr_c`. The atomic task is represented by the tag `BCAtomic_c`. Implementing atomic tasks in the mTask client requires modifying the task tree structure, the procedure to restore after power loss, the rewriter and the interpreter.

We extend task trees with the possibility to store information for atomic tasks. Specifically, three variables: a pointer to the initial inner task tree, a pointer to the active inner task tree, and a boolean indicating whether the atomic task has to be reset. This boolean is related to the power state p in the proposed semantics and is updated when restoring after power loss.

The second modification to the client is in the procedure to restore after losing power. The most obvious adjustment is setting the restart boolean in atomic tasks to true. This indicates that the device restarted after losing power and that the atomic task should be reset. We also modify the restart procedure to expire time-sensitive tasks. The runtime system is unaware of the time passed since losing power, and this time can range widely, depending on the power source of the edge device. As an example, `delay` tasks should expire, because the runtime system is unable to determine whether the delay is finished. Atomic tasks provide developers with a choice in resetting delay tasks. Instead of expiring these tasks, it is also possible to restart the timers by encapsulating them in an atomic task. Listing 3.3 shows two different uses of delay. If power is lost whilst waiting for the delay, (1) causes the timer to expire and (2) restarts the timer with 500 milliseconds. This is an example of how atomic combinators can be used to solve challenges imposed by intermittent computing.

The interpreter is modified to make the task tree node for atomic tasks, and the code in Listing 3.4 shows the procedure to do so. `tt` is a reference to a C structure containing the task tree for the atomic task. `tt->data` is a tagged union that stores specific information for each task type. Atomic tasks can store the initial child task in `atomic.oldtree`, the active child task in `atomic.tree`, and the reboot status in `atomic.reset`. As explained in Section 3.2, the byte code instruction for making the atomic task is preceded by the byte code instruction for making the child task. In Line 1, we access the pointer of the child task from the stack by decreasing the stack pointer `sp` by 1, casting this memory pointer to a task tree pointer, and storing this pointer in `data.atomic.oldtree`. The parent of this initial child node is set to the atomic task tree that we are creating, using `tasktree_set_ptr_tree(...)` in Line 2. Line 3 sets the active tree to `NULL` to have the rewriter make a copy of the initial child task to the active child task. Finally, the reset status of the atomic task is set to `false` in Line 4 to prevent the rewriter from resetting the atomic task right away.

```
(1) delay (lit 500)
(2) atomic (delay (lit 500))
```

Listing 3.3: Different `delay` behaviour after power loss.

```
1 tt->data.atomic.oldtree = (struct TaskTree *)stack[--sp];
2 tasktree_set_ptr_tree(tt->data.atomic.oldtree, tt);
3 tt->data.atomic.tree = NULL;
4 tt->data.atomic.reset = false;
```

Listing 3.4: Interpreter code to make an atomic node.

The final modification to the client is made in the rewriter. The code in Listing 3.5 shows the steps for rewriting the atomic combinator. This code can be split up into four segments: resetting after power loss, cloning the initial child task, rewriting the active child task, and cleaning up when the active child task is stable.

1. Lines 1–6 contain code for resetting the active child node if the rewriting takes place after losing power. Line 2 ensures the active child task is removed, using the context of the current task `current_task` where `stack` indicates the location on the stack where this operation can be performed. The power state, as explained in Section 3.3.2, is set to continue (∘) in Line 4 by changing the boolean to `false`. The final step in the process of resetting the active child is setting the pointer of the active child task to `NULL` in Line 5.

2. The second part, in Lines 8–11, clones the initial child task to the active child task if there is no active child task, either due to a reset or because this is the first rewrite. Cloning the tree is done in Line 9, and this is comparable to copying a value instead of a reference: a new task tree is created independent of the initial task tree. A reference to the current atomic task tree `tt` is passed as the parent of the cloned task tree.

3. The active child task is recursively rewritten in Line 13, and to keep track of the new stack pointer, `sp` is sent and updated with the new stack pointer after rewriting the active child node. As a result, the parent node of the atomic combinator can observe the value of the active child node.

4. The final bit is cleaning up, in Lines 15–19, if the stability of the active child node is *stable*, denoted by `MTStable_c`. If this is the case, there is no need to preserve the atomic combinator as an encapsulation. The work on the active child task is done, and we no longer want this result to be overwritten on power loss. To achieve this, Line 17 marks the initial child task, that is no longer necessary, for deletion and Line 18 replaces the atomic task with the active child task. By cleaning up after a stable task value, we prevent wasting precious memory space, unnecessary rewrites of the atomic combinator, and undoing the progress in the stable child task.

```
1  if (tt->data.atomic.reset) {
2      mem_mark_trash_and_destroy(t->data.atomic.tree,
3          current_task, stack);
4      tt->data.atomic.reset = false;
5      tt->data.atomic.tree = NULL;
6  }
7
8  if (tt->data.atomic.tree == NULL) {
9      tt->data.atomic.tree =
10         tasktree_clone(tt->data.atomic.oldtree, tt);
11 }
12
13 safe_rewrite(tt->data.atomic.tree, stack, stability, sp);
14
15 if (*stability == MTStable_c) {
16     mem_mark_trash_and_destroy(t->data.atomic.oldtree,
17         current_task, stack+sp);
18     mem_node_move(tt, tt->data.atomic.tree);
19 }
```

Listing 3.5: Executed when rewriting an atomic node.

3.4 Evaluating the implementation

The implementation of the atomic combinator is evaluated using the example code in Listing 3.6. This application is based on the example from Figure 1.1 with smart sunshades. Lines 18–24 show the main procedure for this sunshade, where the motor is represented with digital pin 6 and waiting for the motor is simulated with a ten-second delay.

The expected behaviour without power loss is that the `motorPin` is set to `HIGH` for ten seconds after sending a request, before turning `LOW` again until the next request. When power is lost during the ten-second delay, we expect the `motorPin` to turn high before turning low again after ten seconds. This behaviour is tested in a Linux desktop environment where power loss is simulated with a `SIGINT` signal to the client process to kill it. The test shows that the system does, in fact, show the expected behaviour according to the proposed semantics. This verifies that the atomic combinator works as intended in the given example.

```
1  Start w = doTasks (main <<@ ApplyLayout frameCompact) w
2
3  requestShare = sharedStore "request" 0
4
5  main :: Task ()
6  main = enterDevice
7     >>? \spec→withDevice spec deviceTask
8     >>* [ OnAction (Action "Stop") (always (shutDown 0))
9         , OnAction (Action "Reset") (always main) ]
10 where
11   deviceTask :: MTDevice → Task ()
12   deviceTask dev = liftmTask count dev
13                 -|| updateSharedInformation [] requestShare
14
15 count :: Main (MTask v ()) | mtask, lowerSds v
16 count = declarePin D6 PMOutput \motorPin→
17   lowerSds \request=requestShare
18   In fun \processRequest=(\()→
19         getSds request
20     >>*. [IfValue ((==.) (lit 1))
21           \_ → atomic (writeD motorPin true >>|. delay (ms 10000))]
22     >>|. setSds request (lit 0)
23     >>|. writeD motorPin false
24     >>|. processRequest ())
25   In {main=processRequest ()}
```

Listing 3.6: Server code to simulate a smart sunshade in mTask.

3.4.1 Determinism

Tasks in mTask are theoretically deterministic, meaning that the same input results in the same output. This is a desirable property to prevent unexpected behaviour after repeating the same task twice due to power loss. The input to a task contains the state (σ), including peripherals and SDSs. Developers should be aware that updating SDSs in atomic tasks alters the state and, therefore, the input of the atomic task. In addition, reading a peripheral may also change the behaviour of the atomic task. Take an atomic task that increments a counter and then acts upon the value of the counter. The system repeatedly increments the counter when it loses power, affecting the behaviour of the task. Reading a temperature sensor within the atomic task can also affect the behaviour of the task. So, atomic tasks are deterministic, but developers should expect states to change, which leads to different input of a task and consequently different behaviour of the task. Especially when reading from SDSs and peripherals.

3.4.2 Parallelism

The example of Listing 3.6 evaluates the behaviour of the atomic combinator in the context of a sequential composition. In other words, there is no parallelism involved in the example. As explained in Chapter 2, mTask supports multiple tasks to run at the same time. The implementation of the atomic combinator stores a reset state in every atomic task individually, and these reset states are updated when restoring from a power loss. If the client is in a state with two or more parallel atomic tasks, they will all reset the active child node to the initial child node because of the individual state. So, it is safe to use atomic tasks within parallel task compositions.

It is also possible to use parallel compositions within the child task of an atomic task. In the end, all combinators are agnostic of their subtasks, and only the child task's observable task value and its stability matter. When restoring from power loss, the atomic combinator resets the active child task with the initial child task, thus resetting the progress of the parallel combinator in the same way as with a sequential combinator.

3.4.3 Nested atomic tasks

It is possible to use the atomic combinator within the child task of an atomic task: a nested atomic task. This can occur in complex applications with convoluted child tasks, or the atomic combinator can be used as an immediate child of another atomic task. In either case, a nested atomic task serves no purpose. After losing power, the reset boolean is set to true in all atomic tasks, including the nested atomic tasks. However, by definition of the semantics in Section 3.3.2, the first atomic task that is encountered in a composition resets its active child task with the initial child task after a restart. Consequently, the nested atomic tasks are reset, and the boolean is set to false again. So, a nested atomic task affects execution in no way other than requiring an additional step in the rewrite process.

3.4.4 Energy buffer

When an IoT device uses intermittent computing, it could get stuck when a task requires more energy than the physical energy buffer can hold, because the task will constantly restart. This holds for existing tasks and compositions, but this requires particular attention when working with atomic tasks. Atomic tasks are able to grow very large, and it is up to the developer to consider the energy it takes to complete a task and the energy that is available. Take the atomic task in Line 21 of Listing 3.6 as an example. This task takes more than ten seconds to complete. If the energy buffer is only large enough to support nine seconds of work, the client is unable to progress. So, developers should be aware of the physical limitations of the IoT device when working with large atomic tasks.

Chapter 4

Related Work

The concepts behind atomic tasks touch upon established and novel concepts within the field of Computing Science. This chapter covers related properties such as atomicity and idempotency, similar concepts like atomic sections in parallel computing, as well as alternative solutions to atomic tasks.

4.1 Intermittent computing

Implementing intermittent computing can be done in a variety of ways, and this research on atomic tasks is specifically based on the concept of checkpointing in TOP. An alternative solution to checkpointing is using a static task model using redo-logging to restore the volatile state [11]. This solution reduces overhead time and memory consumption as it does not store the entire memory state after a task. However, this static task model is not feasible for mTask, given the limited resources on IoT devices to redo all rewriting steps, as well as the inconsistency of peripheral input. Another alternative to regular checkpointing is proposed in Hibernus [2], where a checkpoint is only made when power is about to be lost. It would be possible to implement this in mTask as well, by measuring a battery's state of charge before making a checkpoint. Checkpointing can be done explicitly or implicitly. Explicit checkpointing requires the developer to define where a checkpoint is made, whilst in implicit checkpointing, this is determined by the compiler or interpreter. The current implementation of intermittent computing in mTask uses implicit checkpointing, as it is well suited for the TOP paradigm [17]. In contrast, the static task model by Maeng et al. [11] requires developers to define where a task begins and ends. Altogether, choosing an implementation for intermittent computing is based on the framework and specific requirements, such as memory usage and the effort for the developer.

4.2 Atomicity

Atomicity is an established property within computing science, and analysing existing applications teaches us about previous approaches. The property of atomicity in the context of software is all about executing a group of statements as if it were just one statement. One application where this property is required is parallel programming, to prevent race conditions in shared-memory parallel programs. Race conditions can occur when the program subsequently reads, modifies, and writes a value. This can be solved using critical sections: blocks of code that are executed in full and without interruption [13]. Another application that requires atomicity is updating databases, where it ensures that transactions are either completed in full or not at all. These transactions can also be used for parallel computing by using transactional memory. A transaction changes shared memory in subsequent steps. The transaction is either completed in full or the shared memory is restored to the state before the transaction [4]. An approach similar to transactional memory is also used in the previously mentioned implementation of intermittent computing [11]. In that example, redo-logging is used on shared resources to commit or abort changes. In conclusion, atomicity is an established property that refers to executing multiple instructions as one, which should be completed either in full or again in its entirety.

4.3 Atomic sections

Research has been done on the topic of atomic sections in intermittent computing before. Although the implementations are built using different programming paradigms, the implementation of atomic sections helps us in better understanding the matter at hand. Research by Majid et al. [12] shows an implementation of dynamic task execution to improve performance for intermittent computing. Coala is a system that accepts static task decompositions and it groups or splits tasks based on the available energy. This means that the developer defines tasks, but when a task is not likely to terminate, Coala makes a partial commit during the task. The system can therefore reboot during a task to improve progress. Coala solves the issue of task atomicity by allowing developers to disable these partial commits for certain tasks. Research by Li and Qiu [7] mentions the waste of time and energy when an atomic task is interrupted by a power loss. The paper describes atomic tasks as tasks that, when interrupted, invalidate all previous work of the task. The proposed algorithm, Psched, schedules tasks based on future energy predictions, criticality and task deadlines. Chain is an older model that uses channels to communicate values between two tasks, and each task is atomic and idempotent, meaning that running the task twice will not alter the result [3]. Chain uses procedural programming, and the

developer defines tasks as any composition of statements. The predominant programming paradigm in existing research in the field of intermittent computing is procedural programming. In our implementation, idempotency is less of an issue because mTask is built on a functional programming language. The exceptions are consulting data from an SDS or reading from peripherals. Additionally, the TOP paradigm allows for an integrated and implicit decomposition of tasks that are atomic by themselves. The knowledge gap that is solved in this research is the addition of a composition of tasks that should be treated atomically, and where power loss invalidates the results of previous tasks in the composition.

4.4 Alternative solution

Currently, there are no alternative solutions that solve the problem where power loss should invalidate the results of previous tasks within certain task compositions. There is a solution that solves a similar problem: the lack of knowledge on how much time has passed since the last power loss. A solution to this problem is timekeeping. Timekeeping prevents using time-sensitive information after a long time interval. Research by Winkel et al. [18] explores the possibilities of timekeeping by discharging a small capacitor over a large resistor. Timekeeping improves progress because information that is still valid would not have to be measured again after losing power. In this sense, timekeeping could be used as an alternative to atomic tasks in the case of time-sensitive information. However, there are use cases, such as user interaction, that require immediate response and would therefore have a deadline of zero milliseconds. In that case, timekeeping as a solution for atomicity would bring significant and unnecessary overhead. The choice between atomic tasks and timekeeping depends on the type of problem that needs to be solved.

Chapter 5

Conclusions

Intermittent computing poses challenges with time-sensitive information and input interaction. Analysing the current implementation of intermittent computing in mTask shows that the client stores checkpoints after rewriting task trees and restores these checkpoints when restarting after losing power. By defining the atomic combinator in the mTask server library, developers can use this atomic combinator to make atomic task compositions. The server sends the instructions for the atomic composition to the client, which interprets these and deals with the semantics. The client resets any progress made within the atomic task when restoring from power loss, according to the proposed semantics of the atomic combinator. The implementation works as intended, and the result is an extension to the mTask framework that lets developers define atomic tasks when using intermittent computing.

5.1 Future work

Two suggestions for further improving intermittent computing in TOP are researching the feasibility of task scheduling and implicit atomic tasks. Task scheduling prevents wasting time and energy on tasks that do not finish in time before losing power, as discussed in Section 4.3. This is particularly useful in combination with atomic tasks, because these compositions can be larger and require more time, hence wasting more resources when the progress is reset. The second suggestion is researching implicit atomic tasks. When the byte code generator or the client's interpreter recognises time-sensitive tasks or user interaction, atomic compositions can automatically be generated. For example, by noticing that certain tasks are unable to become stable.

Acronyms

ASLR Address Space Layout Randomisation. 12

DHT Digital Humidity and Temperature. 5, 6

DSL Domain-Specific Language. 5, 6

IoT Internet of Things. 2–4, 6–8, 11, 12, 20, 21

SDS Shared Data Source. 7, 10, 14, 15, 19, 23

TOP Task-Oriented Programming. 2, 4–6, 8, 14, 21, 23, 24

Bibliography

[1] Elina Antonova. mtask semantics and its comparison to tophat. Bachelor's thesis, Radboud University, 2022. https://www.cs.ru.nl/bachelors-theses/2022/Elina_Antonova___1057069__mTask_semantics_and_its_comparison_to_TopHat.pdf.

[2] Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 7:15–18, 2015.

[3] Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. *SIGPLAN Not.*, 51:514–530, 2016.

[4] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300, 1993.

[5] Paul Hudak. Modular domain specific languages and tools. In *Proceedings. Fifth International Conference on Software Reuse*, ICSR '98, pages 134–142. IEEE, 1998.

[6] Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. A task-based dsl for microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018. Association for Computing Machinery, 2018.

[7] Xuejin Li and Keni Qiu. Psched: An efficient task scheduling approach considering atomic tasks in self-powered systems. In *2024 13th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2024.

[8] Mart Lubbers. *Orchestrating the Internet of Things with Task-Oriented Programming*. Radboud University Press, 2023.

[9] Mart Lubbers and Peter Achten. Clean for haskell programmers. https://arxiv.org/abs/2411.00037, 2024.

[10] Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. Writing internet of things applications with task oriented programming. In Zoltán Porkoláb and Viktória Zsók, editors, *Composability, Comprehensibility and Correctness of Working Software*, pages 3–52. Springer International Publishing, 2023. ISBN 978-3-031-42833-3.

[11] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: intermittent execution without checkpoints. *Proc. ACM Program. Lang.*, 1, 2017.

[12] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Trans. Sen. Netw.*, 16, 2020.

[13] Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1: 74–88, 1992.

[14] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. itasks: executable specifications of interactive work flow systems for the web. *SIGPLAN Not.*, 42:141–152, 2007.

[15] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. Task-oriented programming in a pure functional language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, PPDP '12, pages 195–206. Association for Computing Machinery, 2012.

[16] Tim Steenvoorden. *TopHat. Task-Oriented Programming with Style.* PhD thesis, Radboud University, 2022. https://hdl.handle.net/2066/253701.

[17] Cas Visser. Effortless intermittent computing. Master's thesis, Radboud University, 2025. https://www.cs.ru.nl/masters-theses/2025/C_Visser___Effortless_Intermittent_Computing.pdf.

[18] Jasper de Winkel, Carlo Delle Donne, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. Reliable timekeeping for intermittent computing. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 53–67. Association for Computing Machinery, 2020.