Language Integrated STM in C# Using the Roslyn Compiler

- An Alternative to Locking

Tobias Ugleholdt Hansen
Andreas Pørtner Karlsen
Kasper Breinholt Laurberg

```
class Account
{
    public atomic long Balance { get; set; }

    static void Main()
    {
        var salary = new Account(1000);
        var studentLoan = new Account(-50000);
        TransferMoney(1000, salary, studentLoan);
    }

    public Account(long balance)
    {
        Balance = balance;
    }

    public static void TransferMoney(long amount, Account from, Account to)
    {
        atomic
        {
            from.Subtract(amount);
            to.Add(amount);
        }
    }

    public void Add(long amount)
    {
        atomic
        {
            Balance += amount;
        }
    }

    public void Subtract(long amount)
    {
        atomic
        {
            if (Balance - amount >= 0)
            {
                Balance -= amount;
            }
            else
            {
                throw new Exception(”Insufficient funds!”);
            }
        }
    }
}
```

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg Ø
http://www.cs.aau.dk

Title:
Language Integrated STM in C# Using the Roslyn Compiler - An Alternative to Locking.

Project Period:
Spring Semester 2015

Project Group:
dpt109f15

Participants:
Tobias Ugleholdt Hansen
Andreas Pørtner Karlsen
Kasper Breinholt Laurberg

Supervisor:
Lone Leth Thomsen

Page Numbers: 120 + 4 appendices

Date of Completion:
June 8, 2015

Abstract:

This master thesis investigates whether language integrated STM is a valid alternative to locking in C# in terms of usability, and provides additional benefits compared to library-based STM. To do so, an extension of C# called AC# was implemented. AC# provides integrated support for STM, including conditional synchronization using the retry and orelse constructs, and nesting of transactions. AC# was implemented by extending the open source Roslyn C# compiler. To power AC# a library-based STM system, based on the TLII algorithm, was implemented. The extended compiler transforms AC# source code to regular C# code which utilizes the STM library. For each concurrency approach: AC#, library-based STM and locking in C#, four different concurrency problems, representing different aspects of concurrency, were implemented. These implementations were analyzed according to a set of usability characteristics, facilitating a conclusion upon the usability of language integrated STM. Our evaluation concludes that AC# is a valid alternative to locking, and provides better usability than library-based STM.

Preface

This report documents the master thesis done by group dpt109f15 at the Department of Computer Science at Aalborg University. The thesis was written as part of the Computer Science (IT) study program in the spring of 2015 at the 10th semester.

The first time an acronym is used it will appear in the format: Software Transactional Memory (STM). Inline quotations and names will appear in *italics*. The work presented in this report is based on work or results described in books, articles, video lectures, and research papers from outside sources. The full list of acronyms along with the bibliography, appendix, and summary can be found at the end of the report.

We would like to give a special thanks to our supervisor Lone Leth Thomsen, from the Department of Computer Science, for her excellent guidance and immaculate attention to details. She supplied invaluable help throughout the project with professionalism and black humour. Her feedback has been indispensable and has significantly raised the quality of the final result. Her constructive criticism helped us to narrow down the subject and kept us motivated and enthusiastic about the project.

The report is structured with dependencies between the chapters, and the following can be used as a reading guide:

- Chapter 1 "Introduction" presents the motivation for choosing the topic, the related work, the scope, and the hypothesis. Lastly the method of evaluation is presented.
- Chapter 2 "Background Knowledge" establishes the term "locking", and the key concepts of Software Transactional Memory (STM). This knowledge is required in order to understand the remaining work. If the reader is familiar with these areas, this chapter can be skipped.
- Chapter 3 "Roslyn" outlines the structure of the Roslyn compiler, which enables the integration of STM into C#.
- Chapter 4 "Requirements for AC#" analyze the requirements to the STM system which executes transactions in AC#.
- Chapter 5 "Design and Integration" describes the decisions related to designing and integrating AC# based on the requirements.

- Chapter 6 “STM Implementation” describes the implementation of the STM system that powers AC#, and is based on the requirements and design choices.
- Chapter 7 “Roslyn Extension” describes how Roslyn is extended to encompass language integrated STM, thus being a compiler for AC#.
- Chapter 8 “Evaluation of Characteristics” evaluates AC#, its associated STM library and locking in C# according to the evaluation method described in Section 1.5.
- Based on this evaluation, a conclusion on the hypothesis is made in Chapter 9 “Conclusion”.
- To reflect on the decisions made throughout the report, Chapter 10 “Reflection” discusses the choices made and their consequences.
- Continuation of the work in the future and its potential is discussed in Chapter 11 “Future Work”.

Tobias Ugleholdt Hansen
tuha13@student.aau.dk

Andreas Pørtner Karlsen
akarls13@student.aau.dk

Kasper Breinholt Laurberg
klaurb13@student.aau.dk

Contents

1 Introduction

This chapter describes the motivation behind this project in Section 1.1. Related work is presented in Section 1.2 and the scope of the project is addressed in Section 1.3. The hypothesis and problem statement questions are presented in Section 1.4. Finally, the project evaluation method is stated in Section 1.5.

1.1 Motivation

Today the increase in CPU speed comes in the form of additional cores, as opposed to faster clock speed[1]. In order to utilize this increase in speed, many of the popular sequential programming languages, such as C, C++, Java, and C#, require changes or new additions in order to adapt[2, p. 56]. Our recent study[3], analyzed the runtime performance and characteristics of three different approaches to concurrent programming: Threads & Locks (TL), STM, and the Actor model. The study concluded that STM eliminates many of the risks related to programming in the context of shared-memory concurrency, notably deadlocks, leading to simpler reasoning and improved usability. Additionally, STM fits with the thread model used by sequential languages, and can thereby be applied to existing implementations, without requiring major rewrites. The runtime performance of STM is at a competitive level comparable to that of fine-grained locking[3]. The analysis uncovered one major caveat of STM, it is not orthogonal with side-effects, such as Input/Output (IO) and exceptions[3].

STM has, as of the time of writing, seen only limited official language integration, despite of the advantages STM could provide to sequential languages.To our knowledge STM has only been introduced as an official built-in language feature in Clojure[1] and Haskell[2]. STM has been introduced as a library based solution in sequential languages. However to supply features such as static analysis and syntax support require language integration. From the programmers point of view, a single feature rarely justifies adopting a new programming language. Especially considering that adopting a new language may require rewriting existing code. Thus integrating STM into existing languages will benefit the programmers of these languages.

As of April 2014 and February 2015 Microsoft open sourced the new C# compiler[4] codenamed Roslyn and the Core Common Language Runtime (CLR) [5] respectively. Microsoft thereby facilitates an opportunity to extend the C# language and compiler as well as its runtime system, thereby opening up for integrating STM into the language.

[1]http://clojuredocs.org/

[2]https://wiki.haskell.org/Haskell

1.2 Related Work

This section describes related work within the area of STM. In order to learn from the approaches taken by others as well as achieving a better understanding of the subject, a number of papers, articles, and other research material of relevance has been read. The focus of this section has been on identifying different strategies for language integration of STM as well as different STM implementation strategies.

1.2.1 Composable Memory Transactions

In [6] Harris et al. describe their work with integrating STM into Haskell. Haskell is extended with support for an `atomic` function which takes an STM action as input and produces an IO action as output[6, p. 51]. Evaluating the IO action executes the transaction defined by the atomic function. The Haskell setting allows the authors to divide the world into STM actions and IO actions[6, p. 51], effectively disallowing IO actions within transactions as well as only allowing STM actions to be performed inside transactions. The STM system is implemented as a C library integrated into the Haskell runtime system. The Haskell constructs utilize this library to execute transactions[6, p. 56]. Furthermore, the authors provide a description and implementation of STM constructs for conditional synchronization. The `retry` statement allows a transaction to block until some condition is met at which point the transaction is aborted and re-executed[6, p. 52]. The `orElse` can be used in combination with the retry statement to specify transactional alternatives to be executed in case the previous alternatives encounter a retry[6, p. 52].

1.2.2 STM for Dynamic-sized Data Structures

In [7], Herlihy et al. describe the Dynamic Software Transactional Memory (DSTM) system. DSTM is a library based STM system aimed at the C++ and Java programming languages[7][p. 92]. DSTM uses transactional objects which encapsulate regular objects and provide STM based access and synchronization[7][p. 9]. Each transactional object contains a record of its current value, old value and a reference to the transaction which created the record[7][p. 95]. A Compare-And-Swap (CAS) operation is employed to atomically update the state of a transactional object[7][p. 96]. DSTM is an obstruction-free[8] STM system. Obstruction-freedom guarantees that any thread which runs long enough without encountering a synchronization conflict makes progress[8][p. 1]. Unlike stronger progress guarantees such as lock-freedom and wait-freedom, obstruction-freedom does not prevent livelock[9, p. 47]. As a result DSTM employs a contention manager to ensure progress in practice[7][p. 93]. An extended version of DSTM called DSTM2 is presented in [10]. Here the authors focuses on creating a simple and flexible API for the STM library.

1.2.3 Language Support for Lightweight Transactions

In [11] the authors describe how they integrated STM into the Java programming language by modifying both the compiler[11, p. 4] and virtual machine[11, p. 9]. The authors design, implement and performance test an obstruction free STM system. The STM system uses a non-blocking implementation, and thus guarantees the absence of deadlocks and priority inversion. Additionally, non-conflicting executions are executed concurrently. Per object, it uses an ownership record to track the object's version number as well as which transaction currently owns the object[11, p. 6]. Transaction descriptors are employed in order to keep track of the read and write operations performed by a transaction. Transactions are committed using CAS in order to ensure atomicity[11, p. 7]. The performance tests show how the STM system scales almost as well or better than locking, when the amount of cores available is increased[11, p. 12]. The test cases were performed on data structures, e.g. a ConcurrentHashmap, as opposed to an entire system.

1.2.4 Transactional Locking II

In [12] the TLII STM system designed at Sun Microsystems Laboratories by Dice et al. is described. While many of the other STM systems described here adopt an obstruction-free approach to implementing STM, TLII uses commit time locking[12, p. 199]. A transaction explicitly records its read and write operations in a read and write set[12, p. 198]. Instead of writing directly to memory, all writes are written to the write set. When a transaction is about to commit it acquires the lock on each object in the write set and writes the values contained in the write set to the actual memory locations before releasing the locks[12, p. 200]. This corresponds to a two phase locking scheme[13, p. 455]. A global version clock is used to verify that transactions are executed in isolation[12, p. 201]. As a transaction starts it reads the current value of the global version clock, storing it locally so it can be used for validation. As a transaction is about to commit, it validates its read set by comparing the locally stored read stamp with each object's associated write stamp[12, p. 200]. If any write stamps are higher than the locally stored read stamp, a conflict has occurred and the transaction must abort and re-execute.

1.2.5 A (Brief) Retrospective on Transactional Memory

Inside Microsoft, a group of architects and researchers led an incubation project. Joe Duffy, now director of the Compiler and Language Platform group at Microsoft, gives a retrospective view on their work in [14]. Their goal was to provide a language integrated STM system with support in the Just-In-Time Compilation (JIT) compiler, garbage collector, compiler and debugger. Their overall strategy was to use a version number for optimistic reads, and a lock for writes. Initially they chose weak atomicity and update in-place, but realized that this approach suffered from privatization issues, breaking the isolation. They settled on a write on-commit approach and chose unbounded transactions to provide a broader appeal. Furthermore they relied on compiler optimization through static analysis to remove unnecessary barriers as well as finding violations of the isolation introduced by the programmer. The authors identified STM as a systemic and platform wide technology shift, just like generics. Having a platform wide change, requires careful integration with existing language features, in order to preserve the orthogonality. Several critical operations, that would cause trouble if permitted inside a transaction since their actions are non-reversible, were identified. These include allocation of finalize objects, IO calls, GUI operations, P/Invokes to Win32, library calls and the use of locks. Ultimately, this led to the realization that not all problems are transactional. Very little .NET code, but computations performed solely in memory, could actually run inside transactions. This combined with the privatization issue and several minor but continuous arising problems, caused Joe Duffy to state that the research area of STM was, as of January 2010, not mature enough, and thus STM.NET never made it outside of the incubation project.

1.3 Scope

STM has been an active area of research for almost 20 years[15]. While the research has come far from the initial proposal of statically sized memory transactions, the area still has unsolved problems, including issues with side effects, IO, and exceptions occurring inside transactions[16]. While more research into solving these known problems as well as the creation of new STM algorithms with good performance is of interest to the research community, it is not the focus of this master thesis. Instead the focus is evaluating the integration of STM in the C# programming language. Specifically C# 5.0, the most recent version at the time of writing[17].

The Roslyn compiler project contains compilers for both Visual Basic and C#[4]. As this master thesis focuses on C# we will restrict any investigation into the Roslyn compiler to focus on the C# compiler as well as shared functionality of interest. We realize that the Roslyn compiler contains features for the unreleased C# 6.0, but as these features are uncompleted and not final, they will not be accounted for in the integration.

As of February 2015 Microsoft released the source code for an independent version of its .NET runtime environment Common Language Runtime, called the Core CLR on Github[5]. While this undoubtedly presents many research opportunities for the area of STM as well as other computer science areas, the CLR is considered out of scope for this thesis. The CLR Core is mainly written in C++[5], a language with which we have only limited experience.

Furthermore, the Core CLR consists of roughly 2.6 million lines of code[18] making it a complex and time demanding task to gain an understanding of its structure.

1.4 Problem Statement

The goal of this master thesis is to investigate the usability of language integrated support for STM in C#, compared to a library based solution and existing locking features. To formalize our goal, we have constructed the following hypothesis:

Hypothesis Language integrated STM provides a valid alternative to locking in terms of usability, and provides additional benefits compared to library based STM, when solving concurrency problems in C#.

In order to evaluate the hypothesis, a language integrated STM system for C#, called AC#, and STM library for the .NET platform, will be designed and implemented. Using AC# and the STM library a number of representative concurrent problems will be implemented. These implementations will be compared to equivalent lock based implementations using the evaluation method defined in Section 1.5.

1.4.1 Problem Statement Questions

In order to structure our investigation we have identified a number of problem statement questions. The questions are based on findings from the theory investigated in our previous study[3], investigation of related work, exploratory investigations into STM implementations, and the Roslyn compiler.

1. What features should an STM system for C# contain?
2. What problems exist in integrating STM in C#?
3. What different implementation strategies exist for STM?
4. How is the Roslyn compiler structured?
5. How can the Roslyn compiler be utilized to integrate STM into the C# language?
6. How does the characteristics differ when using locking, library-based STM and language-based STM in the context of C#.

1.5 Evaluation Method

The evaluation is conducted by analyzing characteristics of the different concurrency approaches in a qualitative manner, based on a number of concurrent implementations. The characteristics utilized are an extended version of the characteristics employed in our prior study[3, p. 15-21]. Each of the characteristics highlight key differences in the usability of the concurrency approaches. Combined, they form a comprehensive, although not exhaustive, view of the usability of the concurrency approaches. By evaluating both library-based STM and language based STM their differences will be highlighted which can serve to justify language integration of STM.

1.5.1 Selected Problems

Four concurrency problems have been selected for evaluation:

1. The Dining Philosophers (Appendix B.1)
2. The Santa Claus Problem (Appendix B.2)
3. A Concurrent Queue (Appendix B.3)
4. A Concurrent Hashmap (Appendix B.4)

The Dining Philosophers problem represents a well known concurrency problem which highlights some of the pitfalls associated with synchronization of threads. The Santa Claus problem encompasses a high degree of modeling and requires complex synchronization, e.g. allowing only a predefined number of threads to enter a critical region at a time and waiting on one of multiple conditions. Employing this problem helps investigate what advantages STM provides compared to locking in such scenarios. Both the concurrent queue and hashmap represent real world problems. Concurrent queues are widely used in concurrent applications[19] and concurrent hashmaps can for example be used in a compiler to maintain a symbol table[20]. Both data structures also benefit from fine grained synchronization and is available in a number of languages, including C#. Together these problems provide a varied perspective by exerting different aspects of each approach e.g. waiting on one of multiple conditions and fine grained synchronization.

The source code for each of the implementations can be found in Appendix C, together with a description of the chosen strategy. Common for all the solutions are, that they must solve the problem by using the specified concurrency approach and utilize the strengths of the approach.

1.5.2 Evaluation of Characteristics

For each of the selected problems an implementation will be created using locking, library based STM and AC#. Based on these implementations each concurrency approach will be evaluated according to an extended version of the characteristics defined in our previous work[3, p. 15-21]. These characteristics are a combination of general characteristics for concurrency models such as pessimistic or optimistic concurrency, as well as characteristics such as simplicity and readability which have been used to evaluate the usability of programming languages[21, p. 7]. The additional characteristics are Data Types and Syntax Design, which are relevant as the characteristics are now used to evaluate concrete implementations in a language as opposed to concurrency models in our previous work[3].

Each of these characteristics range from one extreme to another, e.g. high or low readability, however a concurrency approach may not reside at one of these extremes. Therefore each concurrency approach will be given a placement on the spectrum of each characteristic, based on the findings of the evaluation. In order to visualize this placement a scale similar to the one presented in Figure 1.1 is employed. Here X and Y represent the two extremes of the spectrum while the indicators represent the placement of each of the concurrency approaches on the spectrum. As an example X and Y could be low and high writability, Figure 1.1 then shows that each of the concurrency approaches resides more towards the high writability end of the spectrum. As the evaluation of the characteristics is subjective, the placement of each concurrency approach on a spectrum allows for a more clear comparison of the findings in the evaluation.

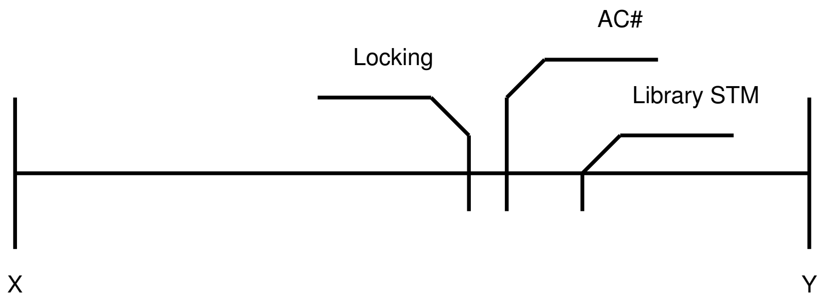


Figure 1.1: Example of characteristic evaluation scale

2 Background Knowledge

This chapter contains background knowledge required to understand the remaining chapters. In Section 2.1 the locking constructs in C# are described, establishing the term "locking" in respect to our hypothesis. Furthermore, the key concepts of STM will be explained in Section 2.2, enabling the reader to understand the details of the design and implementation, discussed in Chapter 5 and Chapter 6 respectively.

2.1 Locking in C#

Locking in C# can in the simplest cases be done with the `lock` statement. For more specialized cases the `Monitor` class can be used. Furthermore, a number of other special case locking constructs can be found in the `System.Threading` namespace e.g. [22] `Mutex`, `SemaphoreSlim`, `SpinLock` and `ReaderWriterLockSlim`. This is not an exhaustive list, but encompasses the constructs used to solve the concurrency problems defined in Section 1.5 as well as some of the more specialized constructs.

2.1.1 Lock Statement

The `lock` statement[17, p. 102] provides a way to acquire and release a lock on a resource. In the scope following the lock statement, a lock is automatically acquired at the beginning and released at the end. The lock provides mutual-exclusion, resulting in threads trying to acquire an already acquired lock, blocking until the lock is released. The `lock` statement ensures that the programmer does not forget to release the lock. Listing 2.1 exemplifies the usage of the lock statement.

Listing 2.1: Lock Statement

```
protected object thisLock = new object();
public int Get() {
  lock(thisLock){
    if (!_queue.IsEmpty()){
      return _queue.Dequeue();
    }else{
      return default(int);
    }
  }
}
```

2.1.2 Monitor

The `lock` statement described in Section 2.1.1 is an assisting way of using the `Monitor` class[23]. The `Monitor` cannot be instantiated as an object, but can be used in any context through its static methods. The methods `Enter` and `Exit` are used to acquire and release a lock on a resource. The `Monitor` class also provides functionality such as allowing the programmer to specify a timeout on the acquisition of a lock. This is done by using the `TryEnter` method. The `Monitor` also facilitates communication between threads with the `Wait`, `Pulse`, and `PulseAll` methods. A thread can call the `Wait` method to release the lock its holding, thus making it possible for other threads to change the state of the resource. When the other thread is done, it can use `Pulse` to notify the next waiting thread of changes, and it will then try to reacquire the lock. `PulseAll` notifies all the waiting threads.

2.1.3 Mutex

The `Mutex` class[24] provides mutual exclusion to a shared resource by allowing only a single thread to acquire the `Mutex` at a time. The `Mutex` ensures thread identity, guaranteeing that only the thread which acquired the `Mutex` can release it. A `Mutex` can be acquired by the method `WaitOne`, which requests ownership of the `Mutex` and blocks the calling thread until the `Mutex` is acquired or a supplied timeout is met. When a thread is done using the `Mutex`, it can use `ReleaseMutex` to release it.

2.1.4 SemaphoreSlim

The `SemaphoreSlim` class[25] is similar to a `Mutex`, but allows multiple threads to enter a shared resource at a time. The amount of threads allowed to enter the semaphore at a given time is set as a constructor argument. Contrary to `Mutex`, the `SemaphoreSlim` does not assure thread identity on the `WaitOne` or `Release` methods. Additionally, there is no guaranteed order in which the waiting threads will enter the `SemaphoreSlim`.

2.1.5 SpinLock

The `SpinLock` struct[26] is similar to the `Monitor` class, and has the methods `Enter`, `TryEnter`, and `Exit` which are called on an instance of the struct. It is implemented as a struct, easing the pressure on the Garbage Collection (GC) but requiring the programmer to pass it by ref. The primary purpose is to allow a thread to spin wait, instead of blocking causing a context switch to occur. This is useful in scenarios where locks are fine grained and in large numbers, or when the holding time of the locks is consistently short. `SpinLock` is not reentrant, consequently if the same thread tries to take the lock twice, an exception will be thrown.

2.1.6 ReaderWriterLockSlim

The class `ReaderWriterLockSlim`[27] is a lock with multiple states allowing threads to differentiate between reading or writing to a shared resource. This enables concurrent reads while keeping writes exclusive, and therefore safe. The methods `EnterReadLock`, `EnterUpgradeableReadLock`, and `EnterWriteLock` enter reading, upgrade and write mode respectively. The read mode is only for reading the shared resource, enabling concurrency with other readers, but mutual exclusion to writers. The upgradeable readlock enables a common pattern where a value is read, and only updated under certain conditions. If a thread acquires a `ReaderWriterLockSlim` in upgrade mode and determines that an update to the protected resource is required, the lock can be upgraded to a write lock without releasing the update lock, thus disallowing other threads from intervening when changing the lock to write mode. The write lock provides mutual exclusion, ensuring exclusive access to the shared resource. All of these methods are also available in a version which allows a timeout to be specified.

2.2 STM Key Concepts

This part contains a modified version of [3, p. 43-48]. The reader can skip this section if she is familiar with STM.

2.2.1 Software Transactional Memory

STM provides programmers with a software based transactional model through a library or compiler interface[28]. It seeks to solve the issues introduced by running multiple processes concurrently in a shared memory space, e.g. race conditions as discussed in [3, p. 22-26]. To handle these challenges, STM offers the programmer ways to define transaction scopes over critical regions. The transactions are executed concurrently and if successful, changes are committed. If a transaction is not successful it will re-executed. Just by defining critical regions, STM ensures atomicity and isolation, and as a result the low level details of synchronization are abstracted away from the programmer[6, p. 48]. Therefore, STM provides a more declarative approach to handle shared-memory concurrency than by using locks. A number of issues related locking discussed in [3, p. 26-30], such as deadlocks, do not exist in STM.

2.2.2 Example of using STM

Languages supporting STM must encompass a language construct for specifying that a section of code should be executed as a transaction and managed by the STM system. This basic language construct is often referred to as the `atomic` block[6, p. 49][11, p. 3]. The `atomic` block allows programmers to specify a transaction scope wherein code should be executed atomically and isolated, as

Listing 2.2: Threadsafe queue

```
1 public int Get() {
2   atomic {
3     if (!_queue.IsEmpty()){
4       return _queue.Dequeue();
5     }else{
6       return default(int);
7     }
8   }
9 }
```

exemplified in Listing 2.2. Exactly how a transaction scope is defined varies between STM implementations. As an example, an STM integrated in the language could look like Listing 2.2 on line 2, while a library based system such as JDASTM[29] uses calls to the methods `startTransaction` and `commitTransaction`.

2.2.3 Conflicts

By declaring an atomic block, the programmer delegates the responsibility of synchronizing concurrent code to the STM system. Avoiding race conditions and deadlocks, while still allowing for optimistic execution introduces conflicts between transactions. In the context of STM a conflict is two transactions perform conflicting operations on the same data, resulting in only one of them being able to continue[9, p. 20]. A conflict arises if one transaction reads the data while the other writes to it. Different techniques of conflict resolution are discussed in [3, p. 45-46 & 52-55]. Despite the different implementation details, the semantics does not change from the point of view of the programmer. However, it is important to know that transactions may conflict, since a high level of contention can negatively affect performance[3, p. 52].

2.2.4 Retry

By enabling the programmer to interact further with the STM system beyond declaring atomic blocks, busy-waiting can be avoided. A common task in concurrent programming is executing code whenever some event occurs. Consider a concurrent queue shared between multiple threads in a producer consumer setup. It is desirable to only have a consumer dequeue an item whenever one is available. Accomplishing this without the need for busy waiting frees the computational resources for other tasks.

In [6] Harris et al. introduce the `retry` statement for assisting in conditional synchronization within the context of STM. The `retry` statement is explicitly placed within an `atomic` block. If a transaction encounters a retry statement during its execution it indicates that the transaction is not yet ready to run and the transaction should be aborted and retried at some later point[9, p. 73]. The transaction is not retried immediately but instead blocks, waiting to be awoken when one of the variables read in the transaction is updated by another transaction[6, p. 51]. By blocking the thread instead of repeatedly checking the condition, busy waiting is avoided.

A transaction using the `retry` statement is shown in Listing 2.3. If the queue is empty the transaction executes the retry statement of line 4, blocking the transaction until it is retired at a later time.

Listing 2.3: Queue with retry

```
1 public int Get() {
2   atomic {
3     if (_queue.IsEmpty()) {
4       retry;
5     }
6     return _queue.Dequeue();
7   }
8 }
```

2.2.5 orElse

In addition to the `retry` statement Harris et al. propose the `orElse` block. The `orElse` block handles the case of waiting on one of many conditions to be true by combining a number of transaction alternatives. These alternatives are evaluated in left-to-right order and only one of the alternatives is committed[6, p. 52]. The `orElse` block works in conjunction with the `retry` statement to determine which alternative to execute. An example of a transaction employing the `orElse` block is shown in Listing 2.4. If an alternative executes without encountering a retry statement it gets to commit and the other alternatives are never executed[9, p. 74]. However, if an alternative encounters a `retry` statement its memory operations are undone and the next alternative in the chain is executed[9, p. 74]. If the last alternative encounters a `retry` statement, the transaction as a whole is blocked awaiting a retry at a later time[9, p. 74].

Listing 2.4: Queue with orElse

```
public int Get() {
  atomic {
    if(_queue.IsEmpty())
      retry;
    return _queue.Dequeue();
  } orElse {
    if(_queue2.IsEmpty())
      retry;
    return _queue2.Dequeue();
  } orElse {
    if(_queue3.IsEmpty())
      retry;
    return _queue3.Dequeue();
  }
}
```

3 Roslyn

This chapter describes the Microsoft Roslyn project, hereafter referred to as Roslyn. Little literature on Roslyn exists. The main source is [30], which is a whitepaper from Microsoft presenting an overview of Roslyn. However the whitepaper's main focus is on the API side of Roslyn, described in Section 3.1, and not the internal details of compilation, which are required in order to integrate STM into the Roslyn C# compiler. Beyond the whitepaper, blog and forum posts have been used, but most of these also only describe the API side. The rest of the knowledge described in this chapter is obtained by inspecting and debugging the source code.

The purpose of this chapter is to obtain the knowledge required to modify the C# compiler with STM support, in order to build AC#. As a result this chapter does not cover the Visual Basic (VB) aspects of Roslyn. The extension of the Roslyn compiler is described in Chapter 7.

In Section 3.1 an introduction to the Roslyn project is given. In Section 3.2 the architecture of Roslyn is described. Following this, Section 3.3 describes the internal details of the compiler phases, information which is valuable in order to select how and where to extend the compiler with STM support. Finally in Section 3.4, a detailed description of the syntax trees of the Roslyn compiler is given.

3.1 Introduction

Project Roslyn is Microsoft's initiative of completely rewriting the C# and VB compilers, using their respective managed language. Roslyn was released as open source at the Microsoft Build Conference 2014[31].

Beyond changing the languages the compilers are written in, Roslyn provides a new approach to compiler interaction and usage. Traditionally a compiler is treated as a black box which receives source code as input and produces object files or assemblies as output[30, p. 3]. During compilation the compiler builds a deep knowledge of the code, which in traditional compilers is unavailable to the programmer and discarded once the compilation is done. This is where Roslyn differs, as it exposes the code analysis of the compiler by providing an API, which allows the programmer to obtain information about the different compilation phases[30, p. 3].

The compiler APIs available are illustrated in Figure 3.1 where each API corresponds to a phase in the compiler pipeline. In the first phase the source code is turned into tokens and parsed according to the language's grammar.

This phase is exposed through an API as a syntax tree. In the second phase declarations, i.e. namespaces and types from code and imported metadata, are analyzed to form named symbols. This phase is exposed as a hierarchical symbol table. In the third phase identifiers in the code are matched to symbols. This phase is exposed as a model which contains the result of the semantic analysis. This model is referred to as a semantic model and exposes methods that answer semantics questions related to the syntax tree for which it is created[30]. Through the semantic model programmers can obtain information such as:

- The type of an expression
- The symbol corresponding to a declaration
- The target of a method invocation

In the last phase, information gathered throughout compilation is used to emit an assembly. This phase is exposed as an API that can be used to produce Common Intermediate Language (CIL) bytecode[30, p. 3-4].

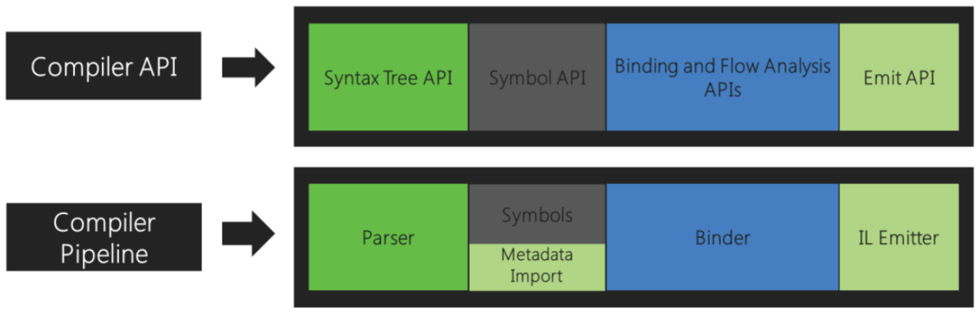


Figure 3.1: Compiler pipeline in contrast to compiler APIs[30, p. 4].

Knowledge obtained through the APIs is valuable in order to create tools that analyze and transform C# or VB code. Furthermore Roslyn allows interactive use of the languages using a Read-Eval-Print Loop (REPL)[32], and embedding of C# and VB in a Domain Specific Language (DSL)[30, p. 3].

3.2 Roslyn Architecture

The Roslyn solution available on github[1], forked[2] on the 9th February 2015, consists of 118 projects which include projects for Visual Studio development, interactive usage of the languages and more as illustrated on Figure 3.2. The

[1]`https://github.com/dotnet/roslyn`
[2]`https://github.com/Felorati/roslyn`

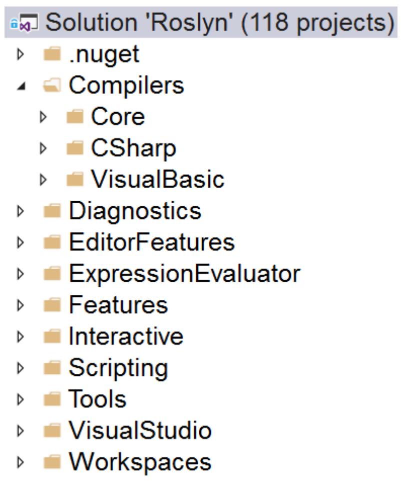


Figure 3.2: Overview of projects in Roslyn solution.

`Compilers` folder contains the source code for the C# and VB compiler, each located in a separate folder. They share common code and functionality located within the `Core` folder, including code for controlling the overall compilation flow. Both compilers use the same patterns for compilation[33, 09:36-10:36].

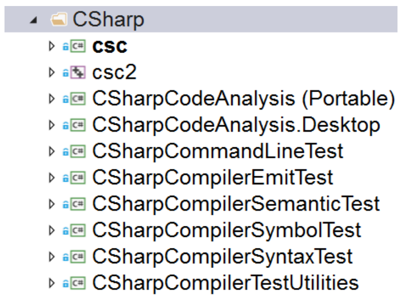

CSharp
csc
csc2
CSharpCodeAnalysis (Portable)
CSharpCodeAnalysis.Desktop
CSharpCommandLineTest
CSharpCompilerEmitTest
CSharpCompilerSemanticTest
CSharpCompilerSymbolTest
CSharpCompilerSyntaxTest
CSharpCompilerTestUtilities

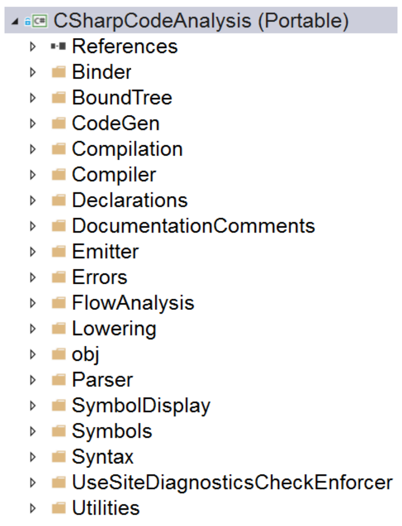
Figure 3.3: Overview of CSharp folder.

The projects contained in the `CSharp` folder are shown on Figure 3.3. The `csc` project is the C# command line compiler, which is the starting point of a C# compilation. The `CSharpCodeAnalysis.Portable` and `CSharpCodeAnalysis.Desktop` projects contain the C# code analysis, which is the actual code required for compilation. The rest of the projects in the `CSharp` folder mainly involve tests for the C# compiler. The `Core` folder has a structure similar to that of the `CSharp` folder, encompassing a `CodeAnalysis.Portable` and `CodeAnalysis.Desktop` project which contain the common core analysis code.

For more information about the architecture, an overview of the Roslyn Compilers call chain can be found in Appendix A.

3.3 Compiler Phases

The C# compiler builds upon concepts from traditional compiler theory, such as lexing, parsing, declaration processing, semantic analysis and code generation[21][34]. Throughout the phases of compilation, traditional concepts such as syntax trees, symbol tables and the visitor pattern[35, p. 366] are also used. This section elaborates on the compiler phases in the compiler pipeline shown in Figure 3.1.



CSharpCodeAnalysis (Portable)
- References
- Binder
- BoundTree
- CodeGen
- Compilation
- Compiler
- Declarations
- DocumentationComments
- Emitter
- Errors
- FlowAnalysis
- Lowering
- obj
- Parser
- SymbolDisplay
- Symbols
- Syntax
- UseSiteDiagnosticsCheckEnforcer
- Utilities

Figure 3.4: Overview of the `CSharp.CSharpCodeAnalysis.Portable` project.

3.3.1 Initial Phase

The initial phase of compilation entails initial work, such as parsing the command line arguments and setting up for compilation, described in further detail in Appendix A. This phase is executed sequentially[36].

3.3.2 First Phase

The first phase involves parsing the source code, which is done in a traditional compiler fashion by lexing source code into tokens and parsing them into a syntax tree, which represents the syntactic structure of the source code. The lexer is implemented using a `switch` which identifies the type of token to lex, given the first character of the token string. The parser is implemented as a top-down parser using the common recursive descent approach. The parsing phase will check for syntax errors in source code, but does not have enough information to check for semantic errors, such as scope or type errors. The phase is concurrent, as several files may be parsed simultaneously[36]. The code for this phase is mainly located within the `Parser` and `Symbols` folder. The syntax tree and its contents are described in more detail in Section 3.4.

3.3.3 Second Phase

The second phase involves creating a `Compilation` type object, specifically a `CSharpCompilation` object. A `Compilation` object contains information necessary for further compilation e.g. all assembly references, compiler options and source code. In the creation of a `CSharpCompilation` object, a declaration table is created which keeps track of type and namespace declarations in source code[36]. This is done sequentially[36]. Additionally the `Compilation` object contains a symbol table, which holds all symbols declared in source code or imported assemblies. Each namespace, type, method, property, field, event, parameter and local variable is represented as a symbol and stored in the symbol table[30, p. 14]. Each type of symbol has its own symbol class, e.g. `MethodSymbol`, which derives from the base `Symbol` class. The symbol table can be accessed by the `GlobalNamespace` property, as a tree of symbols, rooted by the global namespace symbol. Furthermore a range of methods and properties to obtain symbols also exists. The code for this phase is mainly located within the `Declarations` and `Symbols` folders.

3.3.4 Third Phase

In order to enable semantic analysis, the third phase entails fully binding all symbols, which determines what each symbol actually refers to, e.g. what namespace and overloaded method, a particular method refers to. Any problems with symbol binding, like inheritance loops, will be reported. Binding is done concurrently, however binding members of a type will force base types to be bound also. For symbols not related, they can be bound in any order [36].

Additionally, the binding phase also creates a bound tree, which is the Roslyn compilers internal tree used for flow analysis and emitting. In [37] Anthony D. Green states that they do not want to expose the bound tree through the API as:

> *"It has been a long standing design decision not to expose the bound tree. The shape of the bound nodes is actually pretty fragile compared to the shape of the syntax nodes and can be changed around a lot depending on what scenarios need to be addressed. We might store something somewhere one day for performance and remove it the next for compatibility or vice versa"*
>
> – *Anthony D. Green[37]*

Data flow and control flow analysis uses the bound tree to for instance check if statements are reachable, as the C# specification states that an unreachable statement should produce a warning[38]. The code for this phase is mainly located in the `Binder`, `BoundTree`, and `FlowAnalysis` folders.

3.3.5 Final Phase

Finally, in the fourth and final phase all information built up so far is emitted as an assembly. Method bodies are compiled and emitted as CIL concurrently. However methods within the same type are compiled sequentially in a fixed order, typically lexical. The final emitting to the assembly is done sequentially[36]. The code for this phase is mainly located in the `CodeGen` and `Emitter` folders.

3.4 Syntax Trees

Syntax trees are the primary structure used throughout compilation. Syntax trees in the Roslyn compiler have three key attributes[30, p. 6]:

1. A syntax tree is a full fidelity representation of source code, which means that everything in the source code is represented as a node in the syntax tree. If programs are invalid, the syntax tree represents these errors in source code by tokens named `skipped` or `missing` in the tree.
2. A syntax tree produced from parsing must be able to be translated back to the original source code. This is referred to as being roundtripable.
3. Syntax trees are immutable and thread-safe. This enables multiple users to use the same syntax tree in different threads without concurrency issues. As syntax trees are immutable, factory methods exist in order to help create and modify trees. Upon a modification, the factory methods does not copy the entire tree along with the modification, instead the underlying nodes are reused. As a result, trees can be modified fast and with a low memory overhead.

3.4.1 Red And Green Trees

The Roslyn team wanted a primary data structure for compilation with the following characteristics[39]:

- Immutable.
- Form of a tree.
- Cheap access to parent nodes from child nodes.

- The ability to map from a node in the tree to a character offset in the source code.
- The ability to reuse most nodes in the original tree when modifying trees.

However fitting all those characteristics into a single data structure is problematic[39]:

- One problem is simply constructing a tree node, because both the child and parent are immutable and must have a reference to each other, so it is not possible to create one before the other.
- Another problem is reusing nodes for other parents when modifying the tree, as nodes are immutable and it is therefore not possible to change the parent of a node.
- A third problem is inserting a new character into the source code, as the position in source code of all nodes changes after that point. This is makes it problematic to adhere to the characteristic of reusing most nodes when modifying trees, because a modification to source code can change the character offset of many nodes.

Instead the Roslyn compiler uses two types of trees, green trees and red trees, in order to fulfill all their required characteristics.

The green tree is immutable, has the ability upon modification to reuse most unaffected nodes, has no parent references, is built bottom-up, and does not know the absolute positions of nodes in the source code, only their widths[39].

For the expression "5*5+5*5", a typical parse tree is shown in Figure 3.5, and a potential green tree is shown in Figure 3.6. As the green tree nodes do not have parent references and positions in source code, sub-trees and nodes can be reused, which results in a more compact tree. Factory methods are used to create new nodes in the tree in order to determine if existing nodes can be reused or new ones must be created. If nodes for a given expression already exist they are reused, otherwise new nodes are created.

However reuse of existing nodes is not guaranteed, as that requires all nodes to be cached, which according to Vladimir Sadov from Microsoft in [36] makes the reuse caches unnecessarily big. The caches are instead a fixed size, where new nodes will replace older ones when the maximum size is reached. Sadov states that it works pretty well because recently accessed nodes are likely to be accessed in the near future. Another trade-off is that they do not reuse non-terminals with more than 3 children, as it gets more expensive and less likely to match, the more children a non-terminal has[36].

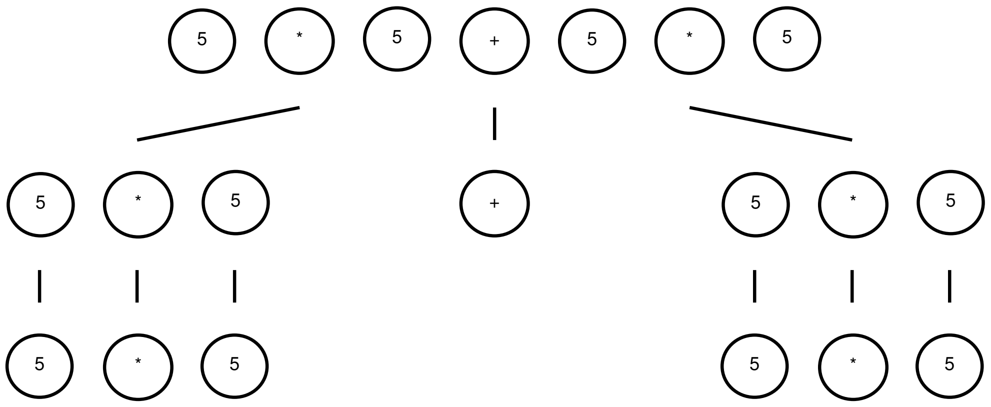


Figure 3.5: Typical parse tree of expression.

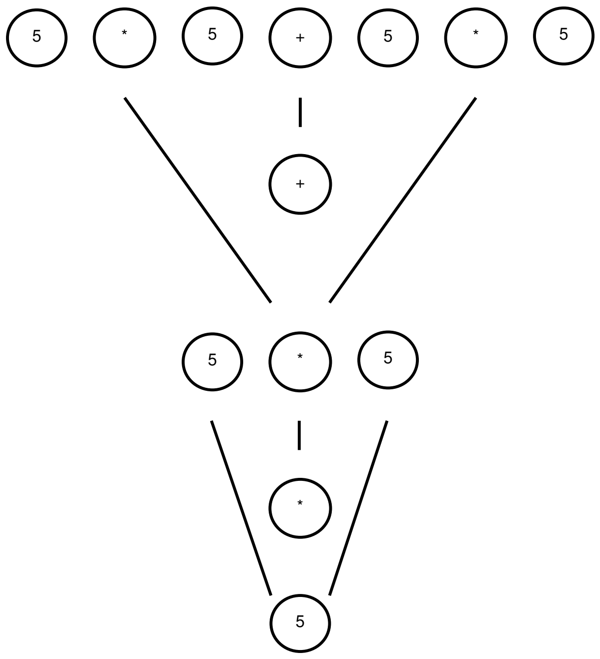


Figure 3.6: Green tree of expression, reusing identical sub-trees. Inspired by [36].

The red tree is an immutable facade built around the green tree. It has parent references and knows the absolute positions of nodes in the source code. However these features prevent nodes from being reused, which means that making modifications to a red tree is expensive. Therefore another approach than building a new red tree upon every modification[39] is used.

The red tree is built lazily using a top-down approach, starting from the root of the tree and descending into children. Once the parent of a node is available, all information required to construct a red node is available. The internal data for the node can be obtained from the corresponding green node. Furthermore the absolute position of the node in source code can be computed, as the position of the parent is known, along with the source code width of all children that come before the given node[36].

So when modifications are made to the source code, an entire new red tree is not computed. Instead the green tree is modified, which is a relatively cheap operation, because most nodes can be reused. In terms of the red tree, a new red node is created as root with 0 as position, null as parent and the root green node as the corresponding green node. The red tree will then only build itself if a user descends into its children and it might only descend into a small fraction of all the nodes in the tree[36].

3.4.2 Contents Of Syntax Trees

The elements contained within syntax trees are syntax nodes, tokens and trivia. In this section these constructs and some of their properties are described. Additionally the supporting constructs Spans, Kinds and Errors are described.

3.4.2.1 Syntax Nodes

Syntax nodes represent non-terminals of the language grammar, such as declarations, statements and expressions. Each syntax node type is represented as a separate class that derives from the base `SyntaxNode` class. As syntax nodes are non-terminals, they always have children, either in the form of other syntax nodes or syntax tokens.

In relation to navigating syntax trees, all syntax nodes has[30, p. 7]:

- A parent property to obtain the parent node
- For each child, a child property to obtain the child
- A `ChildNodes` method to return a list of all child nodes
- Descendant methods i.e. `DescendantNodes`, `DescendantTokens` and `DescendantTrivia`, to obtain a list of all descendant nodes, tokens or trivia for a given node.

Additionally, optional children are allowed, which are represented as `null` if they are not present. For example an `IfStatementSyntax` syntax node has an optional `ElseClauseSyntax` syntax node[30, p. 7].

3.4.2.2 Syntax Tokens

Syntax tokens represent terminals of the language grammar, such as keywords, literals and identifiers. As opposed to syntax nodes, which do not have any children.

Different types of syntax tokens do not have a separate class, instead all syntax tokens are represented by a single `SyntaxToken` type. This means that there is a single structure for all tokens. To get the value of a token, three properties exist: `Text`, `ValueText` and `Value`. The first returns the raw source text as a `String`, including extra characters such as escape characters. The second returns only the value of the token as a `String.` The last returns the value as the actual value type e.g. if the token is an integer literal then the property returns the actual integer. To allow different return types, the return type of the last property is `Object`[30, p. 7-8].

Additionally, for performance reasons the `SyntaxToken` type is defined as a struct[30, p. 7].

3.4.2.3 Syntax Trivia

Syntax trivia represent parts of source code that can appear between any two tokens, such as whitespace and comments. Syntax trivia is not included as a child node in the tree, but instead associated with a given token. A syntax token holds all the following trivia on the same line, up to the next token. Syntax tokens hold trivia in two collections: `LeadingTrivia` and `TrailingTrivia`. The first token holds all leading initial trivia, and the end-of-file token holds the last trailing trivia in source code[30, p. 8].

As trivia are not nodes in the tree, they do not have a `Parent` property. Instead the associated token for some trivia, can be accessed with the `Token` property. Additionally, like syntax tokens, trivia are also structs and have only a single `SyntaxTrivia` type to describe them all.

3.4.2.4 Spans

Every node, token, and trivia knows its position in source code. This is accomplished by the use of a `TextSpan` struct type. A `TextSpan` object holds the start position of a node, token, or trivia in source code and a count of characters, both represented as 32-bit integers[30, p. 8].

Every node, token and trivia has two properties to obtain spans: `Span` and `FullSpan`. The `Span` property includes only the span of the node, token or trivia and not any trivia, where the `FullSpan` property includes the normal span and any leading or trailing trivia.

3.4.2.5 Kinds

Every node, token and trivia has an integer `RawKind` property, used to identify the syntax element type. Each language, C# or VB, contains a `SyntaxKind` enumeration that contains all the nodes, tokens and trivia in the language. The `RawKind` property corresponds to an item in the `SyntaxKind` enumeration for the specific language. The `CSharpSyntaxKind` method gets and automatically cast the bscodeRawKind to an item in the `CSharpSyntaxKind` enumeration[40][30, p. 9].

Kinds are especially important for tokens and trivia, as they have only a single type, `SyntaxToken` and `SyntaxTrivia`. Thus, the only way to identify the particular token or trivia at hand, is by identifying its associated kind.

3.4.2.6 Errors

If programs are invalid as a result of errors in source code, a syntax tree is still produced. These errors are represented as special tokens in the syntax tree, which are added using one of the following techniques[30, p. 9].

1. Insert a missing token in the syntax tree when the parser scans for a particular token but does not find it. The missing token represents the expected token, but it has an empty span and has a true `IsMissing` property.
2. Skip tokens until the parser finds a token from where it can continue the parse. The skipped tokens are added as a trivia node with the `SkippedTokens` kind.

3.4.3 Syntax Tree Generation

The nodes, associated factory methods and visitor pattern for the syntax trees are generated based on the contents of the `Syntax.xml` file located in the `CSharpCodeAnalysis (Portable)` project. The contents of the file describes information such as the fields and base class for each node in the tree. Whenever the compiler is built, a tool which source code resides in the `Tools\CSharpSyntaxGenerator` project is run. It generates the classes for each node defined in `Syntax.xml` along with relevant factory methods, properties for getting the value of each field associated with a node, and factory methods for generating an updated version of the node. The tool also generates the visitor pattern implementation along with the required `accept` methods on each node. Both the red and green tree, described in Section 3.4.1, are generated during this process.

4 Requirements for AC#

This chapter describes the requirements for the STM system powering AC#. The requirements are based on known STM constructs and the characteristics, described in Section 1.5, which the final system must be evaluated upon. The requirements will be used to design and integrate the STM features into C#, described in Chapter 5. Furthermore, the requirements will be used to implement the STM system, described in Chapter 6.

Tracking granularity is described in Section 4.1 followed by a description of the relationship between transactions and variables i.e. what and how variables should be tracked, is addressed in Section 4.2. Section 4.3 describes the choice between strong and weak atomicity. The different types of side-effects and how they are handled is discussed in Section 4.4. Conditional synchronization is described in Section 4.5 and nesting in Section 4.6. Finally Section 4.7 describes opacity before a summary of the requirement is given in Section 4.8.

4.1 Tracking Granularity

A variable tracked by the STM system can be tracked with different granularity: by tracking assignments to the variable, or tracking changes to the object referenced by the variable. These different approaches affect the semantics of the STM system and will be discussed in the subsequent sections.

4.1.1 Tracking of Variables

Tracking changes to the variable directly limits the effect of the STM system to only variables, and not internal changes inside of the referenced object. This is the approach used in the STM system in Clojure[41]. It offers a simple mental model for the programmer, as only changes visible in the transaction scope will be provided with the transactional guarantees of atomicity and isolation. Listing 4.1 shows an example, where the field `_car` is assigned to a new modified car object on line 11. This assignment is tracked by the STM system, as opposed to the example in Listing 4.2, where the internals of the object are not tracked, when a method with a side-effect is called on line 12. The discussion of side-effects is expanded upon in Section 4.4.

This approach can be used in combination with both user defined code and code in contained in binaries, but changes to their internals will not be tracked, and therefore the reusability is limited. Should the programmer want transactional support for the side-effects, the internals must be written to use the STM system.

Listing 4.1: Tracking Assignment to Variables

```
sealed class Car {
  private readonly int _kmDriven;
  public Car(int km) { _kmDriven = km; }
  public Car Drive (int km) {
    return new Car(_kmDriven + km);
  }
}
...
atomic {
  _car = _car.drive(25);
}
```

4.1.2 Tracking of Objects

Tracking of objects allows the STM system to track the internal changes to the fields of an object. This allows the STM system to reverse the modification of an object e.g. when an item is added to a collection inside a transaction and the transaction is aborted, the collection can be restored to the state present before the item was added.

Tracking changes to the internals of an object can be done at the level of only a single object or up to the entire object structure, e.g. the objects referenced by the object etc. In [7] Herlihy et al. present a library based STM called DSTM. DSTM uses an approach where the programmer explicitly has to implement an interface, which allows the STM system to create a shallow copy of an object. The STM system returns a copy of a variable's value to which the transaction can apply its changes. However as the copy is shallow, the internals can reference objects shared with the original. Side-effects on such an object will also affect the original, causing a potentially unintentional breach of isolation. If the programmer want a deeper tracking, she must design the internals using the STM system. The example in Listing 4.2 demonstrates a side-effect on the object contained in the `_car` variable. Because the `_kmDriven` field is assigned a new value, the changes are detected by the STM system. The change to `_engine` is however not detected as the `Car` object is not changed when the `Engine` is degraded.

In [11], Harris et al. present an STM system which tracks changes throughout the entire object structure. Changes to objects are buffered in a log and written to the original if the transaction commits. This deep traceability is enabled by having a part of the STM system in the runtime system, as the entire structure is known at that level, even if an object is from a compiled library. This approach ensures isolation, but requires modifications to the runtime system. In the example of Listing 4.2, both `_kmDriven` and `_engine` will be tracked by STM system presented in [11].

Listing 4.2: Tracking Changes to Object

```
public class Car {
  private int _kmDriven;
  private Engine _engine;
  public void Drive (int km) {
    _kmDriven += km;
    _engine.Degrade(km);
  }
}
...
atomic {
  _car.Drive(25);
}
```

4.1.3 Choice of Tracking Granularity

Deep tracking of objects requires changes to the runtime system which is out of scope for this thesis as described in Section 1.3. This leaves the option of either tracking changes to individual objects, but not the objects internals, or only tracking assignments to variables. Providing support for changes to only individual objects may seem inconsistent from the programmer's point of view, due to changes to referenced objects not being tracked. Therefore tracking of assignments to variables is selected, since it provides a consistent, simple mental model for the programmer, displaying exactly what is tracked by the STM system. The consequence of this choice is that the side-effects cannot be tracked automatically, thus potentially burdening the programmer.

4.2 Transactions & Variables

As described in Section 2.2.2 an STM system must offer some way of defining a transaction scope. As AC# is a language integrated STM system, C# must be extended with syntax for specifying a transaction scope.

An STM system abstracts away many details of how synchronization is achieved. Simply applying transactions over a number of C# variables provides a high level of abstraction but also hides the impact of synchronization. Explicitly marking what variables are to be synchronized can assist the programmer in gauging the performance of a transaction as well as improving the understanding of the execution of transactions, both areas which usability studies[42][43] have found to be problematic for some programmers. C# must be extended with syntax for marking variables for synchronization inside transactions. A variable marked for use in transactions is referred to as a transactional variable. A transactional variable must function similarly to a `volatile` variable. That is, the language must treat a transactional variable as any other variable of the same type when it is utilized. Thus a transactional variable can be passed as an argument into a method with a parameter that is not tracked. Just like a `volatile` variable, a transactional variable must be treated differently than normal variables by the compiler. In terms of usability, these differences must however largely be unnoticeable to the programmer.

Applicability is an important aspect when evaluating the usability of AC#, in comparison to equivalent lock-based and library-based implementations. AC# must not be overly restrictive as this will limit its applicability. Due to this, the STM system must allow reads from transactional variables to occur both inside and outside transactions. For writes to transactional variables a choice exists between allowing and disallowing writes from outside transactions. Disallowing writes from outside transactions will ensure that non-transactional access cannot interfere with transactional access, but it will hamper the usability of the STM system. Allowing writes from outside transactions increases the complexity of the implementation, as any conflicts such writes create must be detected and resolved by the STM system. Since allowing writes from outside transactions provides the best usability, AC# must provide support for such writes in addition to writes from inside transactions. This requirement is closely related to the choice between strong and weak atomicity discussed in Section 4.3.

4.3 Strong or Weak Atomicity

The atomicity guarantee provided by STM systems varies, depending on the semantics provided. In [44] Blundell et al. define two levels of atomicity:

Definition 4.1. *[...] strong atomicity to be a semantics in which transactions execute atomically with respect to both other transactions and non-transactional code.*

Definition 4.2. *[...] weak atomicity to be a semantics in which transactions are atomic only with respect to other transactions.*

Strong atomicity provides non-interference and isolation between transactional and non-transactional code, whereas weak atomicity does not. An example of the differences between strong and weak atomicity is presented in Listing 4.3. Using the `Car` class defined in Listing 4.3, having the `KmDriven` setter called from one thread, while another thread is calling the `Drive` method, strong and weak atomicity yields different results. Under strong atomicity, all changes made inside the `atomic` block on line 12 are isolated from non-transactional code. Additionally, changes made from the setter are isolated from inside the atomic block. The result is that if the setter is called in the middle of the `Drive` method, a conflict will occur which.

If only weak atomicity is guaranteed, given the same scenario, the change made through the setter would be visible inside the atomic block. Thus accessing the same variables from transaction and non-transactional code can lead to race conditions.

Listing 4.3: Level of Atomicity

```
public class Car {
  private int _kmDriven;
  public int KmDriven {
    get {
      return _kmDriven;
    }
    set {
      _kmDriven = value;
    }
  }
  public void Drive (int km) {
    atomic {
      _kmDriven += km;
    }
  }
}
```

4.3.1 Issues with Atomicity Levels

In [9, p. 30-35] Harris et al. summarizes a collection of issues related to the different levels of atomicity. The collection is non-exhaustive, but based on a wide selection of research. The consequence of race conditions can be either:

- Non-repeatable read - if a transaction cannot repeat reading the value of a variable due to changes from non-transactional code in between the readings.
- Intermediate lost update - if a write occurs in the middle of a read-modify-write series done by a transaction, the non-transactional write will be lost, as it comes after the transaction has read the value.
- Intermediate dirty read - if eager updating[3, p. 53] is used, a non-transactional read can see an intermediate value written by a transaction. This transaction might be aborted, leaving the non-transactional code with a dirty read.

The second case is exactly the case described in Listing 4.3, where weak atomicity led to the risk of race conditions between transactional and non-transactional code.

Another issue with weak atomicity is known as the privatization problem. If only one thread can access a variable, the need for tracking it through the STM system ceases, and so does the associated overhead. It is therefore desirable, to privatize a previously shared variable when doing intensive work that does not need to be shared across threads. A technique used for privatizing a variable, `x`, is to use another variable as a shared marker `priv`, which indicates whether or not the `x` is private. This is demonstrated in Listing 4.4. Intuitively one would believe, that if `Thread1` wants to privatize `x`, it can set `priv` to true in a transaction after which `Thread1` has private access to `x`. This is however false, since `Thread2` could read `priv` and assign to `x`, after which `Thread1` executes, setting the values of `priv` and `x`, causing the transaction executed by `Thread2` to abort and rollback. During the rollback the value of `x` is restored to the value it had when `Thread2` wrote to it, causing `Thread1`'s write to `x`, on line 5, outside of the transaction to be overwritten, and lost. This example assumes weak atomicity, commit-time conflict detection, and in place updating[9, p. 34].

Listing 4.4: Privatization Problem

```
 1// Thread 1
 2 atomic {
 3   priv = true;
 4 }
 5 x = 100;
 6 // Thread 2
 7 atomic {
 8   if (!priv) {
 9     x = 200;
10   }
11 }
```

4.3.2 Choice of Atomicity Level

All the issues listed above are related to weak atomicity, and are not present under strong atomicity. Despite of the advantage of strong atomicity, its shortcomings must be considered before choosing. The overhead of guaranteeing atomicity between transactional and non-transactional code can occur a considerable cost[45]. In [45] Spear et al. propose four contracts, for which privatization may be guaranteed. Strong atomicity is ranked as the least restrictive, but comes with a considerable cost. Although the performance is not optimal, Hindman and Grossman show in [46] that strong atomicity with good performance is achievable by source-to-source compiling with optimizations through static analysis.

As described in Section 1.4, the goal of this project is to validate whether STM is a valid alternative to locks and provides additional benefits compared to library-based STM in terms of usability. Therefore strong atomicity in combination with marked transactional variables is chosen for AC# as it provides the best usability.

4.4 Side-effects

Side-effects in methods are a common idiom in C#, and come in different shapes and form. Here side-effects are categorized as in-memory side effects, exceptions or irreversible actions. This section discusses the requirements for handling the different types of side-effects in AC#.

4.4.1 In-memory Side-effects

Side-effects in memory is done by modifying state through references to variables outside of the method scope. An example is Listing 4.2 where the `Drive` method updates the field `_kmDriven` and invokes a method on `_engine`, potentially causing another side-effect. As discussed in Section 4.1, AC# only track assignments to variables, and not changes to the internals of objects. As a consequence, a side-effect such as the one in Listing 4.2 will persist through an aborted transaction. To remedy this, classes must be implemented to track their internals or be immutable. Listing 4.1 shows an immutable implementation of the `Car` class. This design avoids side-effects, and changes to the object will return a new object, which will be tracked if assigned to a transactional variable.

The immutable approach suits STM well, as it is free of side-effects. Additionally it is a less error prone and secure design approach, than mutable objects[47, p. 73]. Microsoft has an official immutable collection package[1], and is therefore giving first class support for immutability. Furthermore Microsoft recommends the use of immutability for "[...] small classes or structs that just encapsulate a set of values (data) and have little or no behaviors"[2]. Guidelines for designing immutable objects can be found in Bloch's Effective Java[47, p. 73-80].

4.4.2 Exceptions

Different approaches exists for handling exceptions raised inside a transaction. AC# applies the programmers intuition of how exceptions work in non-transactional code, to transactional code. Therefore, transactions will not be used as a recovery mechanism as proposed by Tim Harris et al. in [16]. Instead the exception will be propagated if, and only if, the transaction is able to commit at the point where the exception is raised. Otherwise the transactions will be aborted and re-executed. This way, the programmer will

[1] `https://www.nuget.org/packages/Microsoft.Bcl.Immutable`
[2] `https://msdn.microsoft.com/en-us/library/bb384054.aspx`

only receive exceptions from code that actually takes effect, and will be able to recover by catching any exceptions, similar to non-transactional code.

4.4.3 Irreversible Actions

Effects such as IO performed on disk or network, native calls, or GUI operations are not reversible. This makes them unsuitable for use in transactions, since their effect cannot be undone, should the transaction be aborted. In [14], Duffy proposes using a well known strategy from transaction theory[48], having the programmer supply on-commit and on-rollback actions to perform or compensate for the irreversible action. In [16], Harris et al. propose that IO libraries should implement an interface, allowing any IO effects to be buffered until the transactions commit at which point the IO library is given a callback. These solutions either burden the programmer using STM, or the library designer that must implement a special interface.

While the proposed solutions show potential, solving the issue of irreversible actions in transactions is out of scope for this thesis as described in Section 1.3. Due to this, no guarantees are given on the effect of using irreversible actions in transactions, and it is thus discouraged.

4.5 Conditional Synchronization

To be a valid alternative to locking C#, an STM system must be applicable to the same use cases as locking. This requires support for conditional synchronization so that STM can be employed in well known scenarios such as shared buffers and other producer consumer setups[13, p. 128]. Section 2.2 discusses the `retry` and `orElse` constructs proposed in [6] for conditional synchronization and composition of alternatives. Supporting such constructs in C# will increase the applicability of the STM system.

Our previous work in [3] includes an implementation of the k-means clustering algorithm[49, p. 451] in the functional programming language Clojure. Clojure contains a language integrated STM implementation which does not support constructs such as `retry` and `orElse`. As a result the implementation requires the use of condition variables and busy waiting in scenarios where the `retry` construct could have been employed[14]. Supplying `retry` and `orElse` constructs in C# will allow for simpler conditional synchronization without the need for busy waiting, thereby increasing the simplicity and writability in such scenarios.

A disadvantage of providing the `retry` and `orElse` constructs is reduced simplicity. However, as the `retry` and `orElse` constructs are optional, the effects of this disadvantage are reduced. Therefore the conditional synchronization constructs are included in AC#.

4.6 Nesting

The traditional lock-based approach to concurrency has issues with composability due to the threat of deadlocks when composing lock based code[2, p. 58]. STM attempts to mitigate these issues by removing the risk of deadlocks, by allowing transactions to nest. Nesting can occur both lexically and dynamically[50, p. 1][9, p. 42][28, p. 2081].

An STM system for C# must support nesting of transactions as this will allow the system to mitigate one of the major caveats associated with lock based concurrency. A more in depth description of the composability problems of the lock-based concurrency model and nesting of transactions can be found in our prior work [3].

Different semantics exist for nesting of transactions. These are: 1. Flat, 2. Open and 3. Closed[50, p. 1][9, p. 42]. Flat nesting treats any nested transactions as part of the already executing transaction, meaning that an abort of the nested transaction also aborts the enclosing transaction. Closed nested semantics allows nested transactions to abort independently of the enclosing transaction. Under closed nested semantics, commits by nested transactions only propagate any changes to the enclosing transaction, as opposed to the entire system. Open nesting allows nested transactions to commit even if the enclosing transaction aborts and propagates changes made by nested transactions to the entire system whenever a nested transaction commits.

Flat nesting is the simplest to implement, but closed and especially open nesting allows for higher degrees of concurrency[9, p. 43]. Considering the simplicity, readability and level of abstraction provided by the different strategies, as well as the degree of concurrency offered, closed nesting is selected for AC# . In order to improve the orthogonality AC# is required to support both lexical and dynamic nesting.

4.7 Opacity

Opacity is a correctness criteria requiring transactions to only read consistent data throughout their execution[51, p. 1][9, p. 29]. This means that transactions must not read data which will cause them to abort at a later time. Consequently opacity requires that the value read is consistent when the read occurs, but allows the variable to be changed at some later point by another transaction. Transactions must be aborted when reads cannot be guaranteed to be consistent.

By providing opacity, programmers do not have to reason about problems that occur as a result of inconsistent reads[9, p. 28], thereby simplifying the programming model. As an example, consider Listing 4.5

Listing 4.5: Opacity example

```
1  using System.Threading;
2
3  public class Opacity
4  {
5    private atomic static int X = 10;
6    private atomic static int Y = 10;
7
8    public static void Main(string[] args)
9    {
10     var t1 = new Thread(() =>
11     {
12       atomic
13       {
14         X = 20;
15         Y = 20;
16       }
17     });
18
19     var t2 = new Thread(() =>
20     {
21       atomic
22       {
23         var tmpx = X;
24         var tmpy = Y;
25         while (tmpx != tmpy)
26         {
27         }
28       }
29     });
30
31     t1.Start();
32     t2.Start();
33   }
34 }
```

where the two transactional variables `X` and `Y` are defined on lines 5 and 6 as well as the two threads `t1` and `t2` are defined on lines 10 and 19. `t1` simply sets the value of `X` and `Y` as a transaction. `t2` enters a transaction in which it reads the values of `X` and `Y` entering a loop if the values are not equal. Consider the interleaving shown in Figure 4.1. The transaction executed by `t2` reads the value 10 associated with the variable X after which `t1`'s transaction updates the value of both `X` and `Y` to 20. `t2` reads the value 20 associated with `Y.` In an STM system providing opacity, this would not be allowed since the transaction would read inconsistent data. If the STM system does not provide opacity `t2` will enter an infinite loop as `tmpx` and `tmpy` are not equal.

As opacity bolster the simplicity of using STM, it is required for AC#. This will positively impact the readability and writability.

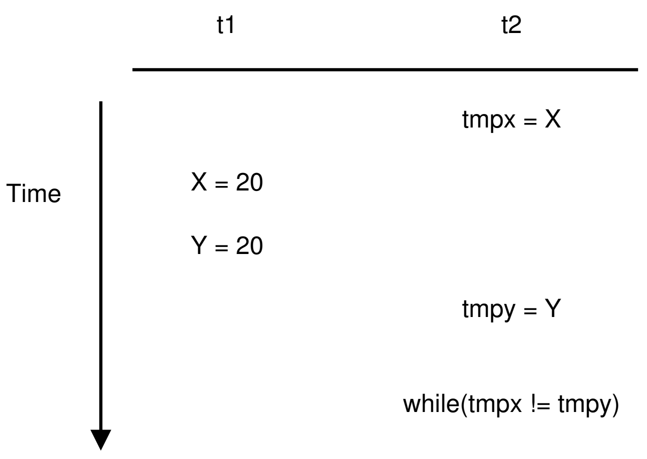


Figure 4.1: Opacity interleaving example

4.8 Summary of Requirements

In the following section, the requirements will be summarized in order to give a clear overview of properties AC# must have.

The granularity of tracking AC# will provide, is on the variable level. That is, the assignments to variables will be tracked, however side-effects to the referenced object will not. Tracking the internals of an object will require the fields of the object to be marked as transactional. A transaction scope can be defined by a syntax extension provided in AC#. Transactions in AC# are executed under the guarantee of strong atomicity, providing isolation between transactional and non-transactional code. As the STM system does not track side-effects, use of immutable objects is promoted. Exceptions occurring inside a transaction will be propagated out of the transaction if the transaction is able to commit. Thus transactions will only be used for synchronization, not recovery of state. All irreversible actions, as mentioned in Section 4.4.3 are discouraged. AC# must facilitate conditional synchronization, by supplying the `retry` and `orElse` constructs. Nesting is allowed in AC# under closed nesting semantics, which strikes a balance between minimizing aborts and ensuring simple semantics for nested transactions. Lastly, opacity will be required for AC#, as this correctness criteria will bolster the simplicity of the STM system.

5 Design and Integration

This chapter describes the considerations and design decisions of AC#, the integration with existing language features, as well as AC#'s syntax & semantics. The design adheres to the requirements defined in Chapter 4, while integrating with existing language features. The decisions made in this chapter will influence the implementation of the STM library described in Chapter 6, and the extension of the Roslyn C# compiler described in Chapter 7.

The atomic block and transactional variables are introduced in Section 5.1. The different kinds of parameters and their integration with transactions is described in Section 5.2. This is followed by an example of the new syntax, demonstrated in Section 5.3. The `retry` and `orelse` keywords for supporting conditional synchronization are introduced in Section 5.4. Section 5.5 shows how AC# supports nesting of transactions. Finally, a summary of the design requirements described in this chapter is given in Section 5.6.

Throughout this chapter, the proposed syntax extensions is described using Extended Backus-Naur Form. The notation uses ? for signaling 0 or 1 occurrences and + for signaling 0 or more occurrences. Parentheses are used for grouping: (item1, item2). Items formatted as `terminal` are terminals, while items formatted as $non-term$ are non-terminals. The syntax extensions presented are based on the syntax used in C# Precisely[52]. This is chosen due to its compact and readable format. As the examples serve to illustrate how the new constructs integrate with the existing language features, some elements of the syntax will not be explained in depth. A reference to where a detailed description can be found will be provided for each example. The complete C# grammar used for parsing can be found at the C# specification[17].

5.1 Transactional Blocks & Variables

As described in Section 4.2 AC# must supply language integrated support for defining transaction scopes as well as declaration of transactional variables. For this purpose AC# extends C# with the `atomic` keyword. The `atomic` keyword will serve as both a statement denoting a transaction scope as well as a modifier applicable to the declaration of variables.

5.1.1 Transactional Blocks

The declaration of a transaction takes the format `atomic` { S }, where S is the general class of C# statements. Any assignments to transactional variables inside an atomic block will be perceived as executed in one atomic step by other transactions and non-transactional code. Return statements inside an atomic block will cause a return from the contained method, just like returns in `if` and `try` blocks.

5.1.2 Transactional Fields

The declaration of a transactional field takes the format:

field-modifiers type `name` = *initializer* (, `name` = *initializer*)+ ;
field-modifiers type `name` (, `name`)+ ;

where *field-modifiers* includes the new `atomic` keyword in addition to the existing modifiers: `static`, `new`, and *access-modifiers*[52, p. 36]. The *field-modifiers* `readonly` and `const` are not included, since their unchangeable nature makes tracking them in an STM system pointless. `volatile` is also left out, as the STM system ensures safety in concurrent contexts.

The *type* can be any existing or user-defined type including `dynamic`. The calculation of *initializer* follows the same rules as standard C# field initializers[52, p. 40].

Listing 5.1 presents an example of atomic field declarations.

Listing 5.1: Local Transactional Variable

```
private atomic string s = "abc";
public atomic static int x, y, z;
```

5.1.3 Transactional Properties

C# facilitates a language feature which eases the encapsulation of fields typically used in Object Oriented Programming (OOP). The feature is known as properties, and can appear in an automatic or a manual form[52, p. 52]. Only the automatic form of properties can be declared as transactional. The format of a transactional property is:

`atomic` *method-modifiers type* `name` { *access-modifier*? `get;` *access-modifier*? `set;` }

method-modifiers include the: `static, new, virtual, override, sealed,` and `abstract` keywords. This is the only change to properties in C#, and the normal rules of property usage still applies, as defined in [52, p. 52].

The manual property lets the programmer specify a backing field manually, allowing him to specialize the functionality of the getter and setter. As a consequence, specifying an `atomic` backing field enables the programmer to implement a transactional property with a specialized getter and setter. The automatic properties generate a backing field automatically, but the getter and setter cannot have specialized functionality.

Atomic automatic properties simplifies the implementation whenever the property serves as a simple getter and setter. An example of such a property is shown in Listing 5.2. If the programmer wishes to supply additional logic for the get and set operations the manual property can be made atomic by accessing an atomic backing field as demonstrated in Listing 5.3. Both cases are frequently used in C#, thus supplying an `atomic` version makes the feature usable in transactions in an orthogonal manner.

Listing 5.2: Automatic Transactional Properties

```
class Car {
  public atomic int KmDriven { get; set; }
}
```

Listing 5.3: Manual Transactional Property

```
class Car {
  private atomic int _kmDriven;
  public int KmDriven {
    get {
      return _kmDriven;
    }
    set {
      _kmDriven = value;
      // Announce value changes
      }
  }
}
```

No restrictions are made on transactional properties marked `virtual`. Consequently the programmer can override a `virtual atomic` property supplying a non-`atomic` implementation. Similarly the programmer is allowed to override a non-`atomic`, `virtual` property with an `atomic` auto implemented property of the same type. This approach is flexible when supplying transactional implementations of existing non-transactional base classes and allows non transactional implementations based on a transactional base class, raising the orthogonality of AC#.

5.1.4 Transactional Local Variables

The declaration of transactional local variables follows the format:

`atomic var name` = *expr*;
`atomic` *type* `name` (= *expr*)? (, `name` (= *expr*)?)+ ;

As with transactional fields, *type* can be any existing or user defined type including the `dynamic` keyword. The deceleration of `atomic const` local variables are not allowed in AC# as const variables cannot be the target of assignments. Listing 5.4 depicts the declaration of two `atomic` local variable in AC#.

Listing 5.4: Local Transactional Variable

```
atomic var s = "abc";
atomic int i = 12;
```

5.2 Transactional Parameters

In C# four different kinds of parameters exist: Value, Reference, Output, and Parameter Array of which `atomic` Value, Reference and Output parameters are available in AC# . This gives the programmer the flexibility to track parameters without requiring additional ceremonial code. The semantics, syntax, and integration of the three parameter types in AC# will be discussed in the following sections. Why `atomic` parameter arrays are not supported is discussed in Section 5.2.4. Transactional parameters adhere to the standard rules in regards to named parameters[17, p. 145]. Just as with the parameter modifiers `ref` and `out`, an overloaded method cannot only differ on the atomic modifier, as it does not change its type. Thus, it follows the rules defined in the specification[17, p. 153-157].

5.2.1 Transactional Value Parameters

According to [17, p. 97]: *"A parameter declared without a ref or out modifier is a value parameter."* If the parameter is a value type, call-by-value semantics is used, thus assignments to the argument have no effect outside the method scope. If the parameter is a reference type, it is also call-by-value semantics, however side-effects that change the referenced object, will persist outside the method scope. Due to the call-by-value semantics, assignments to an argument of reference type will have no effect outside the method scope[52, p. 76].

In AC#, a value parameter can be marked as atomic, thus becoming a transactional value parameter. Semantically a transactional value parameter is equivalent to a transactional local variable which is instantiated to the value of the parameter[52, p. 76]. However, the transactional parameters improve the usability by providing an orthogonal approach for tracking assignments to the parameters in transactions.

For declaration of transactional parameters the format is as follows:

`atomic` *param-modifier*? *type* `name` (= *value* − *initializer*)

where *param-modifier* represents C#'s `ref` and `out` keywords, and *type* can be any existing or user defined type including the `dynamic` keyword, as previously stated. The `atomic` keyword can be combined with optional parameters. An optional parameter declared `atomic` follows the standard rules for optional parameters as defined in[52, p. 46-47].

Listing 5.5 presents an example of an atomic value parameter declaration.

Listing 5.5: Transactional Value Parameter

```
public void Method(atomic int x, atomic string s)
{
  //Work with transactional parameters
}
```

5.2.2 Transactional Reference Parameters

According to [17, p. 97]: *"A parameter declared with a ref modifier is a reference parameter."* As opposed to a value parameter, a reference parameter uses call-by-reference semantics, therefore it is the actual reference used as argument, not a copy. Consequently, assignments to the parameter will take effect outside the method scope, regardless of the parameter being a value or reference type. Assignments to the parameter take effect immediately, both in and outside of the method scope[52, p. 42]. Additionally, a reference parameter is required to be a variable which has been definitely assigned before it is passed as a reference argument[17, p. 97]. Definitely assigned, meaning that the variable is sure to have been given a value at the point where it is used[17, p. 96], in this case passed as a reference parameter.

In AC#, a reference parameter can be marked as atomic, and become a Transactional Reference Parameter (TRP). A TRP differs from a reference parameter by being tracked in transactions. Assignments to a TRP do not take effect outside of the transaction immediately, but when the transaction commits. This is a design choice made to enforce atomicity, as immediate changes to a TRP would enable reading intermediate transaction states. A noteworthy detail is that changes made to a TRP through side-effects will be immediate, since side-effects are not tracked by the STM system, as discussed in Section 4.1.

Listing 5.6 presents an example of a `Point` class declaring two TRP's as parameters for the `ChangeMe` method.

Listing 5.6: Transactional Reference Parameter

```
class Point {
  public atomic int X { get; set; }
  public atomic int Y { get; set; }
  public void CopyMe(atomic ref int x, atomic ref int y) {
    atomic {
      x = this.X;
      y = this.Y;
    }
  }
}
```

5.2.3 Transactional Output Parameters

According to [17, p. 97]: *"A parameter declared with an out modifier is an output parameter."* An output parameter behaves as a reference parameter, except that it does not need to be instantiated before being passed as an argument. Additionally, an output parameter must be definitely assigned whenever the method terminates[52, p. 42]. It is not required that a variable is definitely assigned before being passed as an output parameter, but it is allowed.

In AC#, an output parameter can be marked as atomic, and it becomes a Transactional Output Parameter (TOP). Similar to TRP, a TOP is tracked in transactions, and assignments in transactions take effect outside of the transaction when the transaction commits. The reason for this is the same as for TRPs.

Listing 5.7 presents a modified example of the `Point` class declaring two TOP's as parameters for the `ChangeMe` method.

Listing 5.7: Transactional Output Parameter

```
class Point {
  public atomic int X { get; set; }
  public atomic int Y { get; set; }
  public void CopyMe(atomic out int x, atomic out int y) {
    atomic {
      x = this.X;
      y = this.Y;
    }
  }
}
```

5.2.4 Transactional Parameter Array

According to [17, p. 17]: *"A parameter array permits a variable number of arguments to be passed to a method"*. A parameter array is declared using the `params` modifier. Inside the method, the arguments are represented as an array of length equal to the number of arguments supplied to the `params` parameter. Assignments to the parameter array are possible but this possibility is in our experience rarely used. Additionally, as only assignments to the parameter array are tracked, adding or removing items from the array, would not be tracked in transactions. Thus, the estimated value of adding this feature is close to none. Therefore, a parameter array cannot be marked as atomic in AC#.

5.2.4.1 Transactional Reference & Output Arguments

Four different cases exist for passing variables to `ref` and `out` parameters in AC#. These are:

1. $T \quad \rightarrow \quad$ `atomic` T
2. `atomic` $T \quad \rightarrow \quad T$
3. `atomic` $T \quad \rightarrow \quad$ `atomic` T
4. $T \quad \rightarrow \quad T$

where T is an variable of some type T, `atomic` T is an `atomic` parameter of type T and $T \quad \rightarrow \quad$ `atomic` T describes passing a variable of type T as an argument to an `atomic` parameter of type T. In the first case, a variable reference passed to an `atomic` reference parameter will result in the variable being tracked inside of the method, and assignments to the parameter taking effect outside of the method. In the second case, assignments to the parameter are tracked as if they were assignments to the variable which was passed as `ref` or `out`. In the third case, the variable passed by `ref` or `out` will continue to be tracked inside the method. That is assignments to the parameter are tracked as if they were assignments to the variable passed by bscoderef or `out`. The fourth case follows the standard rules defined in[17, p. 145].

5.3 Example of AC#

Listing 5.8, which is a modified version of the race condition example from our previous study[3, p. 23], presents an example of the syntax extensions AC# brings to C#. On line 6 the transactional field `number` is defined and assigned an initial value of 10. The main method defines and starts two threads. `t1` checks if the value of `number` is equal to 10 and assigns `number` times 3 to the field if the condition is true. These operations are executed as a transaction defined by the `atomic` keyword and associated block on line 11. `t2` assigns 20 to `number` inside a transaction defined on line 17. In this example AC# removes the race conditions associated with `t2` changing the value of `number` between the read on line 12 and the read and write on line 13. In the case that such a change occurs, the STM system of AC# will abort and re-execute one of the implicated transactions in order to resolve the conflict.

Listing 5.8: Transaction Syntax

```
1  using System;
2  using System.Threading;
3  public class RaceCondition
4  {
5    public static atomic int number = 10;
6    public static void Main(string[] args)
7    {
8      Thread t1 = new Thread ( () => {
9        atomic {
10         if (number == 10 )
11           number = number * 3;
12         }
13     });
14     Thread t2 = new Thread( () => {
15       atomic {
16         number = 20;
17       }
18     });
19     t1.Start(); t2.Start();
20     t1.Join(); t2.Join();
21     int result;
22     atomic {
23       result = number;
24     }
25     Console.WriteLine("Result is: " + result);
26   }
27 }
```

As described in Section 4.2 and Section 4.3 AC# must provide strong atomicity as well as allowing reads and writes to occur from outside atomic blocks. As a result, reads and writes occurring from outside transactions will be accounted for when validating transactions. With that in mind, the previous example can be slightly simplified to the example in Listing 5.9. The atomic block on line 17 of Listing 5.8 has been removed, since the `atomic` block contains only a single write. The atomic block on line 24 of Listing 5.8 can also be removed as the read can safely be performed from outside a transaction.

Listing 5.9: Transaction Syntax Simplified

```
1  using System;
2  using System.Threading;
3  public class RaceConditionSimple
4  {
5    public static atomic int number = 10;
6    public static void Main(string[] args)
```

```
  {
    Thread t1 = new Thread ( () => {
      atomic {
        if (number == 10)
          number = number * 3;
      }
    });
    Thread t2 = new Thread( () => {
      number = 20;
    });
    t1.Start(); t2.Start();
    t1.Join(); t2.Join();
    Console.WriteLine("Result is: " + number);
  }
}
```

Allowing access to transactional variables from outside transactions dispenses the need for defining a transaction whenever only a single read or write is to be performed. The atomic block then serves the purpose of combining multiple operations to be executed as a single atomic step.

5.4 Conditional Synchronization

As described in Section 4.5 AC# supports conditional synchronization via the `retry` and `orElse` constructs. AC# extends C# with a `retry` statement that can only be used inside atomic blocks. A retry statement takes the format: `retry`. That is, the keyword is employed as a statement, much like C#'s `break` and `continue` statements[52, p. 102]. Listing 5.10 presents the `Dequeue` method from a transactional queue. The queue is defined over a singly linked list from which items are dequeued from the front and enqueued in the back. If `Dequeue` is called on an empty queue the thread is blocked until the queue is no longer empty. The `Dequeue` method consists of an atomic block, defined on line 11, performing the dequeue operation as a single atomic step. On line 13 the transaction checks if the queue is empty in which case it executes the retry statement on line 14, blocking the transaction to be retried when the `_size` variable changes. If the queue is not empty the head of the queue is removed and the next item in the linked list becomes the new head. Finally the size of the queue is decreased and the value associated with the previous head is returned.

Listing 5.10: Retry Syntax

```
1 using System;
2 public class Queue<T>
3 {
4   private atomic Node _head = null;
5   private atomic Node _tail = null;
6   private atomic int _size = 0;
```

```

  public T Dequeue()
  {
    atomic{

      if (_size == 0)
        retry;

      var oldHead = _head;

      _head = _head.Next;
      if (_head == null)
        _tail = null;

      _size--;

      return oldHead.Value;
    }
  }

...
}
```

Furthermore AC# supports an `orelse` keyword allowing zero to many `orElse` blocks to be associated with an atomic block, much like catch clauses are associated with a `try` statement in many C like languages, including C#[52, p. 96]. The format of the `atomic` block is therefore extended to:

`atomic` { S } (`orelse` { S })+

where S is the general class of C# statements. As an example of the `orelse` construct Listing 5.11 depicts a consumer which extracts an item from one of two buffers via the `ConsumeItem` method. On line 10 the `ConsumeItem` method defines a transaction. If `buffer1` is empty the transaction executes the retry statement on line 12, if not, an item from `buffer` is returned. If `buffer1` is empty and the retry statement is executed, control flows to the `orelse` block defined on line 15. The `orelse` then executes its own transaction returning an item from `buffer2` in case it is not empty. If `buffer2` is empty the retry statement on line 17 is executed, resulting in the entire atomic block blocking until one of the previously read transactional variables change, at which point the transaction is restarted.

Listing 5.11: OrElse Syntax

```
1 using System;
2 public class Consumer<T>
3 {
4   private Buffer<T> _buffer1;
5   private Buffer<T> _buffer2;
6
```

```
  public T ConsumeItem()
  {
    atomic {
      if(_buffer1.Count == 0)
        retry;

        return _buffer1.Get();
    } orelse {
      if(_buffer2.Count == 0)
        retry;

        return _buffer2.Get();
      }
  }
}
```

5.5 Nesting

As described in Section 4.6 AC# supports nesting of transactions under closed nesting semantics. For this purpose AC# allows lexical nesting of atomic blocks as shown in Listing 5.12, where a small program reads two integers from the console. The string arguments are parsed to integers on line 12 to 16. The program then defines two transactions, one starting on line 18 and the other, which is nested inside the first transaction, starting at line 20. The first transaction initiates a nested transaction which sets the transactional variables `X` and `Y` based on the input. Due to the semantics of closed nesting these changes only become visible to the outer transaction when the nested transaction commits. As a result, the outer transaction uses the updated values when it computes the new value of `Z` on line 26, but the remaining system cannot yet see these updated values. When the outer transaction commits, the assignments to `X`, `Y` and `Z` become visible to the rest of the system as a single atomic step.

Listing 5.12: Lexical Nesting

```
1  public class LexicalNesting
2  {
3    private atomic static int X = 0;
4    private atomic static int Y = 0;
5    private atomic static int Z = 0;
6
7    public void Main(string[] args)
8    {
9      if (args.Length != 2)
10       return;
11
12     int tmpx;
13     int tmpy;
14     if (!int.TryParse(args[0], out tmpx)
      || !int.TryParse(args[1], out tmpy))
      return;

    atomic
    {
      atomic
      {
        X = tmpx;
        Y = tmpy;
      }

      Z = X * Y;
    }
    System.Console.WriteLine(Z);
  }
}
```

To increase the usability of AC#, specifically its writability and orthogonality, dynamic nesting of transactions is allowed. As an example, consider Listing 5.13 where a transaction is defined on line 1. The transaction transfers funds from `account1` to `account2` by using the `withdraw` and `deposit` methods, which are themselves defined using transactions. Because of the semantics of closed nesting, the transfer is executed as single atomic step.

Listing 5.13: Dynamically nested transactions

```
1 atomic{
2   var amount = 200;
3   account1.withdraw(amount);
4   account2.deposit(amount);
5 }
```

5.6 Summary of Design

A transactional block can be declared by using the `atomic` statement, to denote a scope for a transaction. Within such a transaction, any assignments to transactional variables, parameters, fields, or properties, will be perceived as executed in one atomic step, by other transactions and non-transactional code.

AC# allows the use of common C# features within a transactional context, including reference and output parameters. To increase the orthogonality of AC# the `atomic` keyword can be used for making a variable, field, property, or parameter as trackable in transactions.

AC# also introduces the keywords `retry` and `orelse`, which can be used for conditional synchronization. `retry` will block the executing thread until a transactional variable previously read by the transactions is changed.

The `orelse` keyword associates an `orElse` block, with an `atomic` block containing a `retry` statement. The `orElse` block will change the initial behavior of the `retry` statement in the `atomic` block. Instead of blocking execution when encountering a `retry` statement, the `orElse` block will be executed. If the last `orElse` block encounters a `retry` the `atomic` block as a whole blocks until one of the previously read values is changed by another transaction.

Additionally, AC# allows nesting of transactions, both lexically and dynamically. Nesting uses closed-nesting semantics, thus inner transactions commits into the outer transaction. The inner and outer transactions are not isolated from each other, thus changes made by the inner transaction are visible to the outer transaction. They are however isolated from all other transactions and non-transactional code.

6 STM Implementation

This chapter describes the implementation of the STM system that executes transactions in AC#. Various implementation algorithms have been proposed for STM of which a few were briefly described in Section 1.2 “Related Work”. This chapter covers the considerations which went into selecting an algorithm for AC# as well as describing the implemented STM library. Specific attention is payed to the requirements and design choices made in Chapter 4 and Chapter 5. Additionally, as the goal of this thesis is to evaluate if language integrated STM is a valid alternative to locking in terms of usability, and provides additional benefits compared to library based STM, building on an existing algorithm is the logical choice, rather than developing a new.

This chapter describes a number of criteria in Section 6.1 that influence the choice of STM algorithm. The final selection of STM algorithm is presented in Section 6.2. The library interface and how the programmer interacts with it is demonstrated in Section 6.3. The internal details of the STM library implementation is described in Section 6.4. Finally, the testing approach used to test the STM library is described in Section 6.5

6.1 Implementation Criteria

This section describes a number of criteria which have an influence on the choice of STM algorithm.

6.1.1 Strong Atomicity

As described in Section 4.3 AC# must support strong atomicity which ensures isolation between transactional and non-transactional code.

Lazy update inherently supports non-transactional reads as the value associated with transactional variables is always the most recently committed value[28, p. 2084][9, p. 21]. Thus a non-transactional read can never read a non-committed value and atomicity is ensured because a single non-transactional read always reads the most recently committed value of a transactional variable. Ensuring atomicity for non-transactional reads is also possible using eager updating, but may incur some overhead in selecting the correct value to read[7].

Although non-transactional writes to transactional variables cannot conflict with one another, they must be tracked by the STM system in order to ensure that any transactions with which they conflict are aborted and retried. In [46] Hindman et al. proposes the concept of mini-transactions. By encapsulating non-transactional access to transactional variables in a transaction, mini-transactions allow the STM system to track their actions. This enables reading from and writing to transactional variables from non-transactional code, while ensuring strong atomicity. Their approach goes even further, supplying optimized non-transactional access which skips much of the unnecessary logging required for normal transactions. As AC# will benefit from using the described approach, the algorithm selected should be capable of supporting it.

6.1.2 Conditional Synchronization

The `retry` construct described in Section 4.5 requires knowledge of which transactional variables have been read, up to the point where a retry is requested[6]. The STM system needs to maintain a set of transactional variables which has been read, for all active transactions, so that a transaction can be blocked until one of these variables change. Such a set is often referred to as a read-set[12][9][53]. STM systems employing lazy conflict detection commonly use such a set to record reads for validation when the transaction is about to commit[12][54]. Choosing an algorithm which already maintains a read-set will limit the overhead of the `retry` statement as well as simplify the implementation as the algorithm must not be modified to support the accumulation of a read-set. Thus selecting an algorithm which inherently supports a read-set is a priority.

6.1.3 Library Implementation

As described in Section 1.4, both a language integrated STM system for C#, in the form of AC#, and an STM library for the .NET platform must be build. Developing a single STM library facilitates reuse as it can be used in both the evaluation and internally in the AC# compiler. The strategy employed must therefore not entail details which require changes to the runtime system, as seen by a number of STM implementations[54][11]. Furthermore the STM library's API must be designed with usability in mind, as this is the focus of the evaluation. However the acSTM library must simultaneously facilitate use from compiler generated code, allowing the extension to the Roslyn compiler described in Chapter 7 to generate code which utilizes the library to execute transactions.

6.1.4 Progress Guarantee

STM algorithms can coarsely be divided into blocking and non-blocking algorithms[9, p. 47]. Blocking algorithms employ some form of locking in order to ensure atomicity, while non-blocking do not[53, p. 59]. Non-blocking algorithms guarantee that the failure of one thread does not keep other threads from making progress[9, p. 47][55, p. 142][53, p. 59]. This makes it impossible to employ locking, as the failure of some thread T, which is holding a lock, will keep other threads from acquiring the lock, and thereby from progressing. Non-blocking algorithms can further be distinguished by the progress guarantees they provide:

- Wait-freedom is the strongest of these, and guarantees that any thread makes progress on its own work if it keeps executing[55, p. 124][53, p. 59].
- Lock-freedom is less strict as it only guarantees that if a thread T keeps executing then some thread (not necessarily T) makes progress[9, p. 47][53, p. 60].
- The least strict progress guarantee is obstruction-freedom[9, p. 47][8][53, p. 61]. As mentioned in Section 1.2, obstruction-freedom guarantees that any thread which runs for long enough without encountering a synchronization conflict makes progress[8, p. 1].

Any wait-free algorithm is also lock-free and any lock-free algorithm is also obstruction-free, but not vice versa[53, p. 60]. While all of these progress guarantees preclude deadlocks, only wait-freedom and lock-freedom preclude livelocks[9, p. 47]. Obstruction free algorithms have been shown to provide efficient implementations in practice[53, p. 61], especially when paired with techniques such as exponential backoff[53, p. 147] and contention managers[9, p. 51].

Whether STM implementations need to be wait-free, lock-free, obstruction-free or even provide any of these guarantees at all is an ongoing discussion in the research community. Arguments have been made for even obstruction-freedom being too strong a guarantee and that STM can be made faster by employing controlled locking[56]. Due to the complexities associated with non-blocking algorithms[57][53, p. 61] and the evidence that blocking STM systems can perform well in practice[12], the initial focus is on blocking STM algorithms. The option of moving to a non-blocking algorithm after producing an initial implementation, as demonstrated by Fernandes et al. in [58] will however be kept open. Due to the limited applicability of C#'s volatile keyword[17, p. 302] ensuring that the most recently written value is always read without resorting to locking can be problematic for the non-supported types. Choosing a locking implementation mitigates this issue.

6.2 Selection of Algorithm

By investigating a number of different STM implementations, the field on which the final selection is based, was narrowed down to the following: McRT[59] developed by Saha et al., TLII[12] developed by Dice et al. and JVSTM[60] developed by Cachopo. McRT and TLII are frequently referenced in the research community and JVSTM is a newer implementation. All of these systems employ locking but in different ways. The algorithms vary greatly in other areas and will provide a broad perspective on possible STM implementations.

6.2.1 McRT

McRT employs encounter time locking and supports two algorithms for synchronization[59, p. 189]. The first is based on reader-writer locks that allows multiple readers to hold the lock simultaneously, while a writer must have exclusive access[59, p. 189]. The second is based on versioned locks where each lock has an associated version number[9, p. 108]. The focus is only on the second approach as this has been shown to have the best performance[59, p. 190].

McRT uses eager updates as this avoids having to check a write-set for a non-committed version of an item, when a read from that item occurs[9, p. 109]. Locking allows the system to guarantee that no transaction reads a value which has not yet been committed[9, p. 108]. Eager updates require the system to maintain an undo list which keeps track of the old values of transactional variables to which a given transaction has written. The undo list is used to revert any items, to which a given transaction has written, to their previous state in case of an abort[59, p. 189]. Eager update makes the commit phase faster as no writes have to be redone[59, p. 190]. The versioned lock algorithm of McRT requires maintaining a read-set which must be validated as part of the commit process. The read-set contains a record of every item read as well as the version number of the item at the time of reading. During the commit phase the read-set of a transaction is validated, ensuring that no read item has a higher version number than the version number present when the item was read, effectively ensuring that no writes have been made to these items while the transaction exectued[59, p. 190].

As McRT uses eager updating, conflicts will arise before transactions are allowed to commit. Therefore McRT detects conflicts only against active transactions[59, p. 189]. Aborting transactions due to a conflict with an active transaction can lead to cases where e.g. a transaction `t1` is aborted due to a conflict with a transaction `t2`, but `t2` is aborted before it can commit due to a conflict with some transaction `t3`[28, p. 2084]. `t1` could then have been allowed to continue as the transaction with which it conflicted never committed.

Conflicts between transactions appear as contention on the locks in the system. If multiple transactions attempt to acquire a lock on the same item a conflict is about to occur. In such cases McRT prefers letting transactions wait for the lock over aborting one of the involved transactions, in order to increase throughput[59, p. 189].

6.2.2 TLII

As briefly described in Section 1.2, TLII only holds locks during the commit process[12, p. 199]. This is known as commit time locking. TLII uses lazy updates where writes are recorded in a write-set and written to the associated memory when the transaction commits. Additionally, any reads performed by a transaction are recorded in a read-set used to validate the transaction before it commits the content of its write-set[12, p. 198]. The accumulation of a read-set for the `retry` will therefore incur no additional overhead.

TLII uses a global version clock to ensure atomicity[12, p. 201]. Transactions record a time-stamp when they start. The time-stamp is used to validate the contents of the read-set before the transaction commits. Each item tracked by the STM system has an associated time-stamp which is updated when a transaction commits a new value. If any item in the read-set has a time-stamp higher than the transactions time-stamp then the item has been modified while the transaction was executing. As a result the transaction must abort and restart.

In order to commit, a transaction must go through the following steps[12, p. 200]:

1. Acquire the lock on each item in the write-set
2. Increment and fetch the version clock
3. Validate the read-set as described above
4. Commit values in the write-set along with the new time-stamps
5. Release the lock on each item in the write-set

The transaction has exclusive access to the items in the write set during the commit phase and detects conflicts lazily against previously committed transactions.

The version clock can be a source of contention as all transactions must increment the version clock as part of the commit operation[9, p. 120]. A number of ways of reducing contention has been proposed [61][62][63].

The base TLII algorithm can lead to false negatives when validating the read-set. If a transaction `t2` executes and commits in between `t1` reading the global clock and reading its first value, `t2` will have incremented the version clock and the time-stamp of `t1` will be invalid before it starts executing its transaction, even though the transaction would be valid in this scenario. In [64] Riegel et al. proposes attempting to update a transactions time-stamp in order to eliminates such false negatives.

6.2.3 JVSTM

JVSTM uses Multiversion Concurrency Control (MVCC)[58, p. 1], as known from databases[65, p. 791]. MVCC requires the STM system to keep a record of the old values for any item that is tracked[65, p. 791]. Reads that would be invalid in a system maintaining only the most recent value for each item, can be redirected to read an older value, allowing the associated transaction to continue[65, p. 791]. As a result JVSTM guarantees that read only transactions will always be able to commit[60, p. 97].

To keep track of the values associated with an item, JVSTM uses what the original designer calls a versioned box[60, p. 63]. A versioned box is an abstraction for an item to be tracked, such as a variable. Each versioned box keeps an ordered sequence of values, called the history of the versioned box. This sequence represents the values that have been committed to the item. Each transaction and version in the history of a versioned box is assigned a version number indicating when the transaction started and when the value was created respectively.

JVSTM uses lazy updates, buffering writes in a transactional local write-set[60, p. 64]. As with TLII, writes are written to the tracked item when the transaction commits. Reads from a versioned box first check if the transactions write-set contains a non-committed value, returning it if that is the case[60, p. 64]. If no such item exists, the history of the versioned box is searched for a suitable value. The value associated with the highest number which is less than or equal to the transactions version number is returned[60, p. 64].

The initial version of JVSTM uses a single global lock to ensure atomicity between committing transactions[60, p. 70], forcing the commit phase to be executed sequentially even though two transactions may not be committing to the same versioned boxes. A lock-free variation of JVSTM, correcting this issue, is proposed in [58].

Keeping the old values for each tracked item takes up additional space and the space consumption increases as new versions are added. Over time some old values might no longer be needed and can therefore be removed. To handle this JVSTM implements a data structure tracking for which number transactions are active and what values where commit for a given transaction number. When a transaction commits it uses the data structure determine whether it has caused any values to be unreachable, cleaning them up if that is the case[60, p. 70][60, p. 88]. This is similar to a garbage collector[34, p. 472].

6.2.4 Final Selection

Based on the criteria presented in Section 6.1, the requirements defined in Chapter 4 and the intended design presented in Chapter 5 it has been decided to base the STM implementation on the TLII algorithm.

TLII uses lazy updating which simplifies the implementation of strong atomicity as a non-transactional read can access a value directly. Additionally the algorithm makes use of a read-set which simplifies the implementation of the `retry` construct. An extension to TLII presented in [9, p. 107] allows support for opacity, which fulfills the requirement presented in Section 4.7. Furthermore the TLII algorithm is well documented from multiple sources, including a library based implementation[12][53, p. 438][9, p. 106].

McRT was discarded as its eager updating strategy based on locking may require non-transactional reads to wait. If an active transaction `t1` has written a value to some variable `x` and a non-transactional read to `x` occurs, the non-transactional read would have to wait for `t1` to finish in order to ensure that the value read is not rolled back at a later point. Additionally conflict detection against active transactions can lead to unnecessary aborts as described in Section 6.2.1.

JVSTM's use of the well known MVCC approach to transactions facilitates a guarantee that read-only transactions will always be able to commit. While this is an attractive property, MVCC also requires the implementation and overhead of garbage collection old values that can no longer be read, thereby complicating the implementation. Additionally, the lock based version of JVSTM uses commit phases which execute sequentially due to global locking. As a result the scalability of the JVSTM algorithm is reduced[60, p. 86].

Performance of the STM system is not the primary concern of this thesis, as described in Section 1.3. However, performance is not neglected completely when choosing an algorithm, as it is an important factor in the context of concurrency. Therefore an algorithm which fulfills the requirements with only minimal performance sacrifices is preferable.

6.3 Library Interface

The STM library is implemented as a C# library, as described in Section 6.1.3. The library interface has been influenced by the design of other STM libraries such as Multiverse[1] and Shielded[2] as well as the introduction to the implementation of STM libraries found in [53].

[1]`https://github.com/pveentjer/Multiverse`

[2]`https://github.com/jbakic/Shielded`

6.3.1 Atomic Blocks

Transactions are started by using one of the overloads of the static `Atomic` method on the `STMSystem` class. The static method `Atomic` is overloaded with support for transactions with and without return values as well as the association of zero to many `orElse` blocks. Listing 6.1 presents an example of transaction declarations. On lines 5-8 a simple transaction declaration corresponding to an `atomic` block is shown. Lines 10-14 depict the declaration of a transaction with a return value. Finally lines 16-23 depict the declaration of a transaction with an associated `orelse` block.

Listing 6.1: Library Transaction Declarations

```
 1 public class TransactionExamples {
 2
 3   public static void AtomicExample
 4   {
 5     STMSystem.Atomic(() =>
 6     {
 7       //Transaction body
 8     });
 9
10     var result = STMSystem.Atomic<int>(() =>
11     {
12       //Transaction body
13       return 1;
14     });
15
16     STMSystem.Atomic(() =>
17     {
18       //Transaction body
19     },
20     () =>
21     {
22       //orelse body
23     });
24   }
25 }
```

6.3.2 Transactional Variables

Transactional variables are created by declaring instances of the generic `TMVar` class. Instances of the `TMVar` class wrap an object to which transactional access is provided. The constructor of the `TMVar` class takes a type parameter specifying the type which the `TMVar` instance wraps.

The `TMVar` class exposes methods and a `Value` property for getting and setting the wrapped object. Any access through the exposed methods and the `Value` property are tracked by the STM system, both transactional and non-transactional. Furthermore the `TMVar` class uses C#'s implicit conversion feature to allow a `TMVar` to implicitly be converted to the wrapped value whenever the `TMVar` occurs as an r-value. Special types, deriving from `TMVar`, are supplied for the common types `int`, `long`, `float`, `double`, `uint`, and `ulong`. These types supply transactional support for common operations such as ++ and -- which are executed on the wrapped object. Listing 6.2 shows the declaration of a number of transactional variables. Lines 5-7 use the generic `TMVar` class to wrap objects of various types. Line 8 declares a `TMInt` , which is the specialized type for transactional integers. Lines 11-17 declares a transaction that on line 13 checks if the boolean value wrapped by `tmBool` is true and the string value wrapped by `tmString` is equal to the string `"abc"`. `tmBool` and `tmString` are implicitly converted to the wrapped value while preserving transactional access, allowing the transactional variables to be used as if it was an instances of the wrapped type. Line 15 uses the ++ operator supported by the `TMInt` type.

Listing 6.2: Library Transactional Variable

```
1  public class TransactionExamples {
2
3    public static void TMVarExample()
4    {
5      TMVar<string> tmString = new TMVar<string>("abc");
6      TMVar<bool> tmBool = new TMVar<bool>();
7      TMVar<Person> tmPerson = new TMVar<Person>(new Person("Bo
           Hansen", 57));
8      TMInt tmInt = new TMInt(12);
9
10
11     STMSystem.Atomic(() =>
12     {
13       if (tmBool && tmString == "abc")
14       {
15         tmInt++;
16       }
17     });
18   }
19 }
```

6.3.3 Retry

The library supports the `retry` construct via the static `Retry` method on the `STMSystem` class. The retry method can be called from both in and outside transactions but will only have an effect when called inside transactions. Listing 6.3 presents an example of how the `Retry` method can be used. Lines 5-11 show a transaction dequeuing an item from a shared buffer. On line 8 the `Retry` method is used to initiate a retry in case the queue is empty. The call on line 15 has no effect as it is not enclosed in a transaction.

Listing 6.3: Library `Retry`

```
public class TransactionExamples {

  public static void RetryExample(Queue<Person> buffer)
  {
    Person result = STMSystem.Atomic(() =>
    {
      if (buffer.Count == 0)
        STMSystem.Retry(); //Initiates retry

      return buffer.Dequeue();
    });

    //Work with the result

    STMSystem.Retry(); //Has no effect
  }
}
```

6.3.4 Nesting

The STM library supports both lexical and dynamic nesting of transactions. Lexical nesting is achieved by specifying a transaction within the body of another transaction, while dynamic nesting is achieved by calling a method, which is defined using transactions, within the body of another transaction. Nesting is done under the semantics of closed nesting as described in Section 4.6. An example of nesting using the STM library is shown in Listing 6.4. Line 6-15 show a lexically nested transaction, while line 20-24 show two dynamically nested transactions as the `Enqueue` and `Dequeue` methods are themselves defined using a transaction.

Listing 6.4: Library Nesting

```
1  public class TransactionExamples {
2
3    public static void NestingExample()
4    {
5      TMVar<string> s = new TMVar<string>(string.Empty);
6      var result = STMSystem.Atomic(() =>
7      {
8        s.Value = "abc";
9        STMSystem.Atomic(() =>
10       {
11         s.Value = s + "def";
12       });
13
14       return s.Value;
15     });
16
```

```
    Queue<Person> buffer1 = new Queue<Person>();
    Queue<Person> buffer2 = new Queue<Person>();

    STMSystem.Atomic(() =>
    {
      var item = buffer1.Dequeue();
      buffer2.Enqueue(item);
    });
  }
}
```

6.4 Internal Details

This section highlights the defining aspects of the STM libraries internal implementation.

6.4.1 Writing a Value

As described in Section 6.2.2 TLII uses a write-set to buffer any writes to be written to the actual locations when the transaction commits. Therefore any writes to transactional variables must be redirected to the write-set instead of the actual location. Listing 6.5 shows how this is accomplished in the STM library. Line 3 gets the currently executing transaction from threadlocal storage. That is, each thread gets its own transaction instance when accessing the `LocalTransaction` property. Based on the status of the transaction, different actions are taken. Each thread accesses a transaction that has the status `Committed` whenever no transaction is currently executing on that specific thread. Therefore the case on line 6 covers writing a value to a transactional variable from outside a transaction scope. How this is accomplished is explained in Section 6.4.4. The case on line 9 covers writing a value as part of a transaction. Here the value to be written is put into the write-set of the transaction performing the write. When the given transaction commits, the value will be written back to the transactional variable.

6.4.2 Reading a Value

How the value of a transactional variable is read depends on the context in which the read occurs. Reading the value of a transactional variable from non-transactional code amounts to reading the value contained within the variable, as this value is guaranteed to be the most recently committed value. On the other hand, a read from transactional code has to read an uncommitted value which was previously written by the same transaction, if such a value exists. Only if no such value is present should the transaction read the value currently contained in the transactional variable. Listing 6.6 shows how reading the value of a transactional variable is implemented in the STM library. The case

Listing 6.5: Writing to a Transactional Variable

```
private void SetValueInternal(T value)
{
  var me = Transaction.LocalTransaction;
  switch (me.Status)
  {
    case Transaction.TransactionStatus.Committed:
      SetValueNonTransactional(value);
      break;
    case Transaction.TransactionStatus.Active:
      me.WriteSet.Put(this, value);
      break;
  }
}
```

on line 6 corresponds to a non-transactional read and are forwarded directly to the base class in order to retrieve the value of the transactional variable. The case on line 8 corresponds to a transactional read. If the transaction's write-set does not contain a value for the current `TMVar` object, lines 10-11 reads the value directly. Otherwise line 13 gets the value, which was previously written by the current transaction, from the current transaction's write-set. Finally, on line 15, the transactional variable is added to the active transaction's read-set, so that the read can be validated before the transaction commits.

Listing 6.6: Reading a Transactional Variable

```
 1 private T GetValueInternal()
 2 {
 3   var me = Transaction.LocalTransaction;
 4   switch (me.Status)
 5   {
 6     case Transaction.TransactionStatus.Committed:
 7       return base.GetValue();
 8     case Transaction.TransactionStatus.Active:
 9       T value;
10       if (!me.WriteSet.Contains(this))
11         value = base.GetValue();
12       else
13         value = (T)me.WriteSet.Get(this);
14
15       me.ReadSet.Add(this);
16       return value;
17   }
18 }
```

6.4.3 Committing a Transaction

Committing a transaction follows the steps described in Section 6.2.2. Listing 6.7 shows the implementations of these steps. The `Commit` method declared on line 1 drives the commit process and returns `true` if the commit succeeds. If the transaction is not nested within another transaction, the body of the if statement on line 3 is entered. The call to the `Validate` method on line 6 acquires the lock on each object in the transaction's write-set, validates the read-set and increments the version-clock acquiring a new writestamp. The `Validate` method returns true if the read-set could be validated correctly and returns the new writestamp using an `out` parameter. All locks are acquired using a timeout in order to prevent deadlocks and all acquired locks are released if validation of the read-set fails. If validation succeeds, the `HandleCommit` method commits all buffered writes to their respective transactional variables, after which all acquired locks are released.

If the transaction is nested within another transaction the else case on line 11 is executed. Nested transactions do not need to acquire the lock on items in their write-set or increment the version clock as they will not be responsible for committing any written values under closed nested semantics. Therefore, a validation of the read-set, as seen on line 13, is sufficient. If the validation succeeds the transaction merges its read and write sets with those of the outer transaction, in order for the outer transaction to commit any written values when it reaches its commit stage. On line 17 the outer transaction is restored as the currently active transaction so that it can be resumed. Finally on line 20 the status of the transaction is changed to committed, before returning on line 21.

6.4.4 Providing Strong Atomicity

As described in Chapter 4 the STM system must support strong atomicity. In Section 6.4.1 and Section 6.4.2 it is described how the STM system is able to detect whether or not read and write operations occur inside or outside a transaction and take different actions based on this information. In the case of a non-transactional read the system reads the most recently committed value as described in Section 6.4.2, ensuring that no uncommitted values can be read. A non-transactional write is more complicated as the STM system must ensure atomicty between both non-transactional access and transactional access. To that extent the approach described by Hindman et al. in [46] of executing non-transactional access as optimized mini transaction is adopted. As no value is read during a non-transactional write, a non-transactional write in the context of the TLII algorithm consist only of (1) Acquiring the lock on the transactional variable to which the write occurs, (2) Incrementing and getting the value of the version clock, (3) Committing the new value and timestamp before, (4) Releasing the acquired lock. Therefore any access to the read-set can be optimized away. The STM library implements this approach in the `SetValueNonTransactional` method of the `TMVar<T>` class shown in Listing 6.8.

Listing 6.7: Committing a Transaction

```
public bool Commit()
{
  if (!IsNested)
  {
    int writeStamp;
    if (!Validate(out writeStamp))
      return false;

    HandleCommit(writeStamp);
  }
  else
  {
    if (!ValidateReadset())
      return false;

    MergeWithParent();
    Transaction.LocalTransaction = Parent;
  }

  Status = TransactionStatus.Committed;
  return true;
}
```

Listing 6.8: Non-transactional Write

```
private void SetValueNonTransactional(T value)
{
  Lock();
  Commit(value, VersionClock.IncrementClock());
  Unlock();
}
```

6.4.5 Providing Opacity

As described in Section 4.7 providing opacity requires ensuring that transactions do not read invalid data throughout their execution. Therefore a transaction must validate a read when it occurs, ensuring that the transaction is not allowed to continue when the read is inconsistent. For this purpose the STM library uses the approach shown by [9, p. 117] and extends the reading of values to the implementation depicted in Listing 6.9. As seen on lines 12-16, reading a value directly from the transactional variable has been extended with validation that ensures the transaction aborts if the variable has been changed since the transaction recorded its timestamp. The timestamp of the transactional variable prior to reading it is stored locally on line 12. After the variable has been read the locally cached timestamp is compared to the current timestamp of the transactional variable and the timestamp of the

Listing 6.9: Providing Opacity

```
1  private T GetValueInternal()
2  {
3    var me = Transaction.LocalTransaction;
4    switch (me.Status)
5    {
6      case Transaction.TransactionStatus.Committed:
7        return base.GetValue();
8      case Transaction.TransactionStatus.Active:
9        T value;
10       if (!me.WriteSet.Contains(this))
11       {
12         var preStamp = TimeStamp;
13         value = base.GetValue();
14
15         if (preStamp != TimeStamp ||  me.ReadStamp < preStamp)
16           throw new STMAbortException("Aborted due to
                 inconsistent read");
17       }
18       else
19         value = (T)me.WriteSet.Get(this);
20
21       me.ReadSet.Add(this);
22       return value;
23   }
24 }
```

current transaction. If the timestamp was changed during the read or the cached timestamp is higher than the timestamp of the current transaction, the current transaction is aborted by throwing a `STMAbortException` on line 16.

6.5 Testing

In order to ensure that the STM library executes transactions according to the requirements defined in Chapter 4, a number of tests have been created.

An improvisational approach for testing is to create some ad hoc throw-away code[66, Chap. 9] and manually interacting with the program, to ensure that code works correctly. This approach has the advantage that tests can be created quickly and errors found quickly. However the approach has the disadvantages that it is unstructured and the testing can not be rerun automatically, so if a later change damages a feature which has previously been tested as working correctly, this may not be discovered.

Another more structured approach is to use unit testing[67], where tests are structured into unit tests, which only focus on the correctness of a small part of the program. Furthermore, unit testing is often automated, so they can easily be rerun to ensure the unit they test still works correctly. A drawback with this approach is that it is more time consuming to setup than adhoc testing.

As unit testing will allow for automatic testing of new features not breaking existing functionality, unit testing has been selected for testing the STM library. The defined tests cover areas such as the execution of transactions, nesting, retry, orelse, strong atomicity and exceptions in transactions. Additionally, a number of tests based on a transactional queue and hashmap implementation has been created. Where the first set of tests cover smaller parts of the STM library, the second set of tests cover the STM library as a whole, ensuring that the library can be utilized for real world scenarios.

In addition a number of tests, which attempt to produce a result which can only occur when a race condition is present, has been created. While such tests can ultimately not prove that no race conditions are present, they do provide a degree of certainty that this is the case. Listing 6.10 shows the `RaceTest1` method conducting a test for possible race conditions. `RaceTest1Internal` is called 10,000 times producing a result for each iteration. Line 7 asserts that each execution of `RaceTest1Internal` does not produce a result which can only occur given that a race condition is present. `RaceTest1Internal` executes a read modify write operation on one thread, while another thread writes to the same variable. The call to `Thread.Yield` on line 21 signals the underlying system that `t1` can be descheduled in favor of other threads, allowing `t2` to assign `result` a new value after `t1` has read the value of result, but before it has computed and assigned the new value. `Thread.Yield` does not cause `t1` to be desheduled in all cases, only when the underlying scheduling determines that it is appropriate.

Listing 6.10: RaceTest1

```
[TestMethod]
public void RaceTest1()
{
  for (int i = 0; i < 10000; i++)
  {
    var result = RaceTest1Internal();
    Assert.IsTrue(result != 120);
  }
}

private int RaceTest1Internal()
{
  var result = new TMVar<int>(10);

  var t1 =  new Task(() =>
  {
    STMSystem.Atomic(() =>
    {
      if (result.Value == 10)
      {
        Thread.Yield();
        result.SetValue(result.Value * 10);
      }

      return result.GetValue();
    });
  });

  var t2 = new Task(() => STMSystem.Atomic(() =>
  {
    result.Value = 12;
  }));

  t1.Start();
  t2.Start();
  t1.Wait();
  t2.Wait();

  return result.Value;
}
```

7 Roslyn Extension

This chapter describes how the Roslyn C# compiler is extended to support the constructs of AC# described in Chapter 5. Section 7.1 describes the overall extension strategy. In Section 7.2 the changes made to the lexer and parser are described. Following this, Section 7.3 presents examples of the transformations made on the syntax tree by the extension while Section 7.4 describes the testing approach employed to test the Roslyn extension. Finally, Section 7.5 describes areas where the initial prototype implementation, described in this chapter, conflicts with the intended STM design described in Chapter 5.

7.1 Extension Strategy

The Roslyn C# compiler is extended by modifying the lexing and parsing phases with support for the language constructs described in Chapter 5. The extended parsing phase outputs an extended syntax tree containing direct representations of the language features provided by AC#. The syntax tree is then analyzed to identify AC# constructs, followed by a transformation where the language extension of AC# is transformed into equivalent C# code which utilizes the STM library described in Chapter 6. This syntax tree is then passed to the remaining C# compiler phases, utilizing the compilers semantic analysis and code generation implementations. The approach is visualized in Figure 7.1, which is a modified version of Figure 3.1. The transformation phase utilizes both the extended syntax tree and symbol information gathered through the Roslyn API. By doing modifications in the early phases, the amount of changes required is minimized, as the rest of the phases can be reused without modifications. Furthermore, modifications are done on the stable syntax tree, rather than the unstable bound tree, as described in Section 3.3.

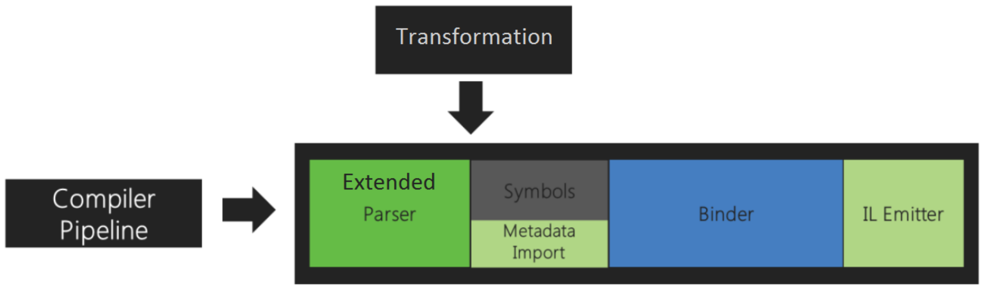


Figure 7.1: Extension occurs at the syntax tree and symbol level. The parser is extended to output an extended syntax tree and transformation of this tree occurs before the binding and IL emission phases.

The described approach was selected due to the following reasons:

1. As described in Section 3.3.2 the C# lexer uses a simple `switch` case to identify tokens, which chooses the type of token to identify based on the first character of the token, while the parser uses a recursive descent technique. Both the lexer and parser are implemented by hand. The techniques employed have a low degree of complexity, as the first method uses a simple `switch` and the latter method corresponds to parsing one non-terminal at a time. Thus modifying the lexer and parser is simpler than if more complex techniques had been employed, such as Look Ahead LR (LALR) parsing[68].

2. The Roslyn compiler generates the nodes composing its syntax trees along with factory methods and a visitor pattern implementation based on the content of an Extensible Markup Language (XML) file. Therefore adding new nodes to represent the extended language constructs, such as the `atomic` block and `retry` statement, simply amounts to adding definitions for these nodes to the XML, allowing the employed tool to generate source code for the new nodes.

3. As described in Section 3.4 Roslyn's syntax trees are designed to be fast to modify by reusing underlying green nodes instead of creating complete copies[30, p. 6]. This speaks for conducting transformation on the level of the syntax tree despite its immutable implementation.

4. The Roslyn project has been designed to allow programmers to utilize the information gathered during the compilation phases by analyzing the syntax tree, the information in the symbol table, and the results of semantics analysis. These parts of the compiler are exposed as a public API allowing access to both syntactic and semantic information. Utilizing this API during the transformation phase allows the transformation to draw on the existing semantic analysis to answer questions such as, what method is the target of a method invocation, without the need for implementing complex analysis.

5. By parsing an extended syntax tree and transforming it into a regular C# syntax tree, the existing semantic analysis and code emission implementations can be utilized.

Despite of the many advantages of the selected approach, a number of disadvantages also exists. The following disadvantages has been identified:

1. By modifying the syntax tree, the roundtripable property, described in Section 3.4, is lost, as the syntax tree no longer represents the original

source code. Consequently, any error or warning generated by the compiler refers to the transformed source code, which the programmer does not see, as opposed to the original AC# source code. This requires our own code analysis, to give meaningful errors attached to the original source code. Alternatively, the transformation could be performed at a later compilation phase, e.g. modifying the bound tree in the binding phase or do changes before emitting code in the emit phase. This would preserve the roundtripable property, but also limit the reuse of the existing compiler.

2. As `atomic` local variables are translated into variables of a corresponding STM type, the types of these need to be known. C# allows the programmer to utilize type inference for local variables using the `var` keyword. Consequently the type of a local variable is not required to be defined in the source code. To remedy this, the extension must infer the type before translation. Roslyn offers the possibility to evaluate the type of an expression, which is used to infer the type of `atomic` local variables with unknown types. Consequently, the extension must perform some of the work that the Roslyn compiler does at later stages of the compilation, reducing the reuse of the later compilation phases.

7.2 Lexing & Parsing Phases

This section describes the changes conducted in order to extend the lexing and parsing phases of the Roslyn C# compiler to support the constructs described in Chapter 5.

7.2.1 Lexer Changes

As described in Chapter 5, AC# introduces three new keywords: `atomic,` `orelse` and `retry`. Consequently the lexer has been extended to identify these keywords as tokens of the correct kind. The C# lexer initially lexes keyword tokens as if they are identifiers. If an identifier token corresponds to a keyword, a keyword token with the correct kind is returned instead.

In order to identify the new keywords, their definition has been added to the lookup methods of the `SyntaxKindFacts` class. This class defines the `GetKeywordKind` which the lexer uses to identify the keyword kind of an identifier, if the identifier represents a keyword. Additionally, the `SyntaxKindFacts` class defines the `GetText` method for determining the string representation of a keyword based on its kind.

To allow tokens to represent the new keywords, the `AtomicKeyword,` `OrelseKeyword,` and `RetryKeyword` entries has been added to the `SyntaxKind` enumeration. As described in Section 3.4.2.5, the `SyntaxKind` enumeration contains an entry for each type of node, token, or trivia in C#. Whenever the lexer identifies an occurrence of one of the new keywords, a token with the corresponding kind is returned. For example an occurrence of the `atomic` keyword results in a token with the kind `AtomicKeyword` being returned.

7.2.2 Syntax Tree Extension

The design described in Chapter 5 adds the `atomic`, `orelse` and `retry` constructs which the existing syntax tree cannot express. Therefore the syntax tree must be extended to support these language constructs. As described in Section 3.4.3 nodes composing the syntax tree, along with factory methods and the visitor pattern implementation, are generated on the basis of an XML file. Adding additional nodes to the syntax tree therefore amounts to defining them in the XML notation, which has been done for the three previously mentioned constructs. Listing 7.1 shows the XML code defining the `AtomicStatementSyntax` node which represents an `atomic` block in the syntax tree. Line 1 defines the name and base class of the node while line 2 defines its kind. The `AtomicStatement` kind has been added to the `SyntaxKind` enumeration as described previously. Line 3 defines a property on the node which holds the token representing the `AtomicKeyword` which starts the definition of the atomic block. The property has a constraint specifying the kind of the token, that can be associated with the property which is defined on line 4 as well as a comment given on lines 5-9. Line 11 defines a statement property which holds the block of statements associated with the defined `atomic` block. Line 18 defines a property containing a `SyntaxList` of `orelse` blocks associated with the `atomic` block. This relationship has been modeled after the relationship between a C# `try` statement and its `catch` clauses, as both have a zero to many association. Finally lines 25-27 defines a comment for the `AtomicStatementSyntax`, while line 28-30 defines a comment for the factory method.

Listing 7.1: AtomicStatement XML definition

```
1  <Node Name="AtomicStatementSyntax" Base="StatementSyntax">
2    <Kind Name="AtomicStatement"/>
3    <Field Name="AtomicKeyword" Type="SyntaxToken">
4      <Kind Name="AtomicKeyword"/>
5      <PropertyComment>
6        <summary>
7          Gets a SyntaxToken that represents the atomic keyword.
8        </summary>
9      </PropertyComment>
10   </Field>
11   <Field Name="Statement" Type="StatementSyntax">
12     <PropertyComment>
13       <summary>
```

```
        Gets a StatementSyntax that represents the statement to
            be executed when the condition is true.
      </summary>
    </PropertyComment>
  </Field>
  <Field Name="Orelses" Type="SyntaxList&lt;OrelseSyntax&gt;">
    <PropertyComment>
      <summary>
        Gets a SyntaxList containing the orelse blocks associated
            with the atomic statement.
      </summary>
    </PropertyComment>
  </Field>
  <TypeComment>
    <summary>Represents an atomic block.</summary>
  </TypeComment>
  <FactoryComment>
    <summary>Creates an AtomicStatementSyntax node</summary>
  </FactoryComment>
</Node>
```

Declaration of transactional local variables, fields and parameters does not require modifications to the syntax tree as the standard nodes for these constructs allow a collection of modifiers to be associated with the declaration. Any atomic modifiers are simply added to the collection along with any other modifiers.

7.2.3 Parser Changes

As described in Section 3.3.2 the C# parser uses a recursive descent strategy implemented by hand. As customary for recursive descent implementations, each non-terminal has a method responsible for parsing that particular non-terminal. For the new `atomic`, `orelse` and `retry` such a method has been added. Furthermore, the methods for parsing local variables, fields, parameters, and properties have been modified to allow for an atomic modifier, as well as generating error messages for any unsupported modifier combinations such as `atomic const` and `readonly const`, as defined in Section 5.1.2. Errors are associated with the erroneous nodes as is customary for the Roslyn compiler. A later compilation phase generates error messages and cancels code emission if any errors are present in the syntax tree.

Listing 7.2 shows the `ParseAtomicBlock` method responsible for parsing an `atomic` block. Line 3 parses an `atomic` keyword by returning a token representing the keyword while line 4 parses the block of statements representing the transaction body. On line 6 to 21 any `orelse` blocks associated with the `atomic` statement are parsed , if any are present. Line 7 allocates a `SyntaxListBuilder.` The `Allocate` method reuses existing space if possible, in order to limit the overhead of allocation. As seen on line 12 the parsing of the actual `orelse` block is delegated to its corresponding method. On line 20 the space allocated by the call on line 7 is freed. The `try finally` construct ensures that the space is freed in case of an exception. Finally on line 23 the syntax factory is used to create the `AtomicStatementSyntax` that is returned.

Listing 7.2: Method for parsing `atomic` block

```
 1 private StatementSyntax ParseAtomicBlock()
 2 {
 3   var atomicKeyword = this.EatToken(SyntaxKind.AtomicKeyword);
 4   var block = this.ParseEmbeddedStatement(false);
 5
 6   var orelses = SyntaxFactory.List<OrelseSyntax>();
 7   var orelseBuilder = _pool.Allocate<OrelseSyntax>();
 8   try
 9   {
10     while (this.CurrentToken.Kind == SyntaxKind.OrelseKeyword)
11     {
12       var clause = ParseOrelse();
13       orelseBuilder.Add(clause);
14     }
15
16     orelses = orelseBuilder.ToList();
17   }
18   finally
19   {
20     _pool.Free(orelseBuilder);
21   }
22
23   return _syntaxFactory.AtomicStatement(atomicKeyword, block,
        orelses);
24 }
```

7.2.4 Symbol Changes

The symbols representing fields, local variables, and parameters have been modified with a new `IsAtomic` property of type `bool`, indicating if the symbol represent an atomic variation of these constructs. The logic which creates the symbol table has further been modified to, for each of these constructs, determine whether the declaration is atomic and set the `IsAtomic` to the appropriate value.

For each usage of a field, local variable or parameter, the semantic model allows for the retrieval of a symbol, representing the corresponding declaration. Based on the added `IsAtomic` property this symbol can be used to determine whether the usage represents the usage of an atomic variable. For cases where only access to atomic variables are of interest, in relation to the STM system, the symbol extension allows for easy distinguishing between access to atomic variables or non-atomic variables.

7.3 Syntax Tree Transformations

This section presents the syntax tree transformations performed during the compilation process. In order to prevent errors due to ambiguity between type names, the transformation process uses fully qualified names[17, p. 73] for any types in the STM library. In the examples presented in this section the simple names have been used, in order to improve readability.

7.3.1 Atomic Block

In Section 5.1.1 the design for the `atomic` block is described. Listing 7.3 depicts the syntax of the atomic block before transformation.

The transformation of an atomic block is done using the following four steps:

1. Construct a lambda expression with a body equal to that of the `atomic` block.
2. Construct lambda expressions for any `orelse` blocks associated with the `atomic` block, with bodies equal to that of their respective original definition.
3. Construct a syntax node for the invocation of the `STMSystem.Atomic` method, supplied with the created lambda expressions as arguments.
4. Replace the `atomic` block with the invocation of the `STMSystem.Atomic` method.

For syntax shown in Listing 7.3 the transformation produces the output shown in Listing 7.4.

Listing 7.3: `atomic` Block Before Transformation

```
atomic
{
  //Block
}
orelse
{
  //Orelse block
}
```

Listing 7.4: `atomic` Block After Transformation

```
STMSystem.Atomic(() => {
  //Block
},
() => {
  //Orelse block
});
```

As a consequence of translating an atomic block into a lambda expression, return statements inside the lambda expression do not return out of the atomic block as described in Section 5.1.1. In order to ensure the wanted semantics, an analysis is performed to identify all atomic blocks containing return statements. In such a case, a return statement is added before the method invocation. Return statements inside nested transactions must return out of the enclosing method. To accommodate this, the analysis also identifies nested transactions.

7.3.2 Field Types

In Section 5.1.2 the design of transactional fields are described. Any field declared `atomic` must have its type substituted to the corresponding STM type in order for the STM system to track any changes to the variable. If a specialized type exist, such as `TMInt` for `int`, then that type is used. Otherwise the generic `TMVar` is used. As the STM types act as wrapper objects that allows the STM system to track how the wrapped values are accessed, all atomic fields must be initialized to an instance of a `STM` type, as accessing the wrapped value will otherwise cause a `NullReferenceException`. If an initializer expression is given as part of the field declaration, the constructor of the wrapping STM object is given the expression as an argument, initializing the wrapped value to the value computed by the initializer expression. If no initializer expression is given, the wrapped value is initialized to the default value for the wrapped type by instantiating the STM object using its parameterless constructor. The transformation of an atomic field declaration follows the steps described below:

1. Determine the type of the wrapping STM object, based on the type of the field declaration.
2. For each variable declared as part of the field declaration, construct an object instantiation expression following the approach described above. This expression serves as the new initializer for the particular variable it was created for.
3. Construct a new field declaration with the same access modifiers and variable names as the original declaration, but substituting the type with the STM object type, and initializer expressions with the created object instantiation expressions.
4. Replace the original field declaration with the constructed field declaration.

Listing 7.5 presents an example of two `atomic` field declarations before transformation while Listing 7.6 shows the result of applying the transformation. Line 3 of Listing 7.5 is transformed to line 3 of Listing 7.6. The type of the field is changed to the specialized integer STM type `TMInt` and the initializer expression is used to initialize the value of the created `TMInt` object. Line 4 of Listing 7.5 is transformed to line 4 of Listing 7.6. The type is transformed to `TMVar<string>`. For `field3`, which original definition does not contain an initializer expression, an initializer expression has been created following the previously described procedure.

Listing 7.5: `atomic` Field Before Transformation

```
1 public class AtomicFieldExample
2 {
3   private atomic int field1 = 1;
4   public atomic string field2 = "Hello world!", field3;
5 }
```

Listing 7.6: `atomic` Field After Transformation

```
1 public class AtomicFieldExample
2 {
3   private TMInt field1 = new TMInt (1);
4   public TMVar<string> field2 = new TMVar<string>("Hello
        world!"), field3 = new TMVar<string>();
5 }
```

7.3.3 Properties

In Section 5.1.3 the design for transactional properties was described. In order to provide the wanted semantics for the automatic form, the transformation involves two parts: an atomic backing field, and a manual property. The transformation takes the following approach for each transactional property identified in the syntax tree:

1. Construct a get-body with the access-modifier of the original property's get. The get-body contains a block that returns the value of the backing field. The backing field is not yet constructed, but a method is used to determine its future identifier to ensure a correct reference.

2. Construct a set-body with the access-modifiers of the original property's set-body. The set-body contains a block where an assignment of `value` is made to the backing field.

3. Construct a manual property with the access modifier of the original property and the get-body and set-body constructed earlier.

4. Construct a private transactional field of the same type as the original property, used as backing field.

5. Insert the new property after the original property and replace the original property with the private transactional field.

Transactional properties are transformed before transactional fields. This enables the transformation of transactional properties to simply generate a backing field which is declared atomic and rely on the transformation of transactional fields to transform the type of the field to the correct STM type. No transformation has to be done for the manual form, as the transactional field used for backing the property is processed as described in Section 7.3.2. The automatic form before transformation is exemplified in Listing 7.7, where the transformation result is shown in Listing 7.8.

Listing 7.7: `atomic` Property Before Transformation

```
class Car {
    public atomic int KmDriven { get; set; }
}
```

Listing 7.8: `atomic` Property After Transformation

```
class Car {
  private atomic int _kmDriven;
  public int KmDriven {
      get {
          return _kmDriven;
      }
      set {
          _kmDriven = value;
      }
  }
}
```

7.3.4 Local Variables

In Section 5.1.4 the design for transactional local variables is described. Similar to fields modified with the `atomic` keyword, `atomic` local variables must have its type substituted to a corresponding STM type. Thus, the approach is similar to the one described in Section 7.3.2, with the exception that a local variable can be declared without specifying the type by using the `var` keyword, relying on compile-time type inference to determine the type. Since the type has not yet been determined at the point of the transformation, Roslyn's API is utilized to infer the type, and replace the `var` keyword with the type. This is done by identifying all local declaration statements with the type `var` and the `atomic` modifier.

The transformation is exemplified with a before example on Listing 7.9, where on line 5 a local variable is declared by using the `var` keyword. The r-value of the statement is an expression, which type can be identified using the Roslyn compiler's API. In Listing 7.10 on line 5 the result shows that the type was infered to a `string`, thus the `var` is replaced by a `TMVar<string>`.

Listing 7.9: `atomic` Local Variables Before Transformation

```
public class AtomicVarExample
{
  public void Method()
  {
    atomic var variable1 = "Hello World!";
    atomic int variable2 = 42;
  }
}
```

Listing 7.10: `atomic` Local Variables After Transformation

```
1 public class AtomicVarExample
2 {
3   public void Method()
4   {
5     TMVar<string> variable1 = new TMVar<string>("Hello World!");
6     TMInt variable2 = new TMInt(42);
7   }
8 }
```

7.3.5 Accessing Transactional Variables

As described in Section 6.3.2 setting the value of a transactional variable is done through the `Value` property of the supplied STM types. As the type of any field, local variable or parameter declared `atomic` in AC# is changed to the corresponding STM type, any assignment must go through the `Value` property. Additionally, the STM types supply support for implicit conversion to the wrapped value, when appearing as r-values. The transformation can however not rely on implicit conversion in all cases as, for example the comparison of two `TMInt` objects results in reference comparison instead of the expected integer comparison. As such, also the r-value appearances of any transactional field, local variable and parameter must be transformed to access the `Value` property, which ensures that the wrapped object is accessed instead of the STM object. Special handling is given to the `++` and `--` operators as the numeric STM types supply transactional implementations of these operators. The transformation ensures that these operators can be used on a transactional variable directly, instead of on its `Value` property.

The transformation of access to transactional variables is divided into two parts. The first deals with the usage of a transactional variable occurring as a single identifier such as `someTMVar`. The second part handles member access expressions such as `object.tmfield1.tmfield2`. While these two implementations differ in the syntactic constructs they work on, their overall approach both follow the steps described below:

1. Identify each usage of a transactional field, local variable, or parameter including both r- and l-value occurrences.
2. Construct a member access expression that accesses the `Value` property of the identified variable.
3. Replace the usage of the variable with the constructed member access expression.

Listing 7.11 presents an example of accessing both a transactional field, local variable and parameter. Listing 7.12 shows the result of applying the transformation. The assignment on line 8 of Listing 7.11 is transformed to access the `Value` property of both its left and right side, as both of the variables involved are transactional. The result of the transformation is shown on line 8 of Listing 7.12. The member access expression on line 10 of Listing 7.11 is likewise transformed to access the `Value` property of both transactional fields involved. The resulting code is shown on line 10 of Listing 7.12.

Listing 7.11: Usage of `atomic` Variables Before Transformation

```
1  public class AtomicExample
2  {
3    public atomic AtomicExample aField;
4
5    public AtomicExample ExampleMethod(atomic int i)
6    {
7      atomic int k = 0;
8      k = i;
9
10     return aField.aField;
11   }
12 }
```

Listing 7.12: Usage of `atomic` Variables After Transformation

```
1  public class AtomicExample
2  {
3    public TMVar<AtomicExample> aField = new TMVar<AtomicExample>();
4
5    public AtomicExample ExampleMethod(TMInt i)
6    {
7      TMInt k = new TMInt(0);
8      k.Value = i.Value;
9
10     return aField.Value.aField.Value;
11   }
12 }
```

7.3.6 Parameters

As with transactional local variables and transactional fields, the type of a parameter declared atomic must be changed to the corresponding STM type in order for the STM system to track assignments to it. Listing 7.13 presents a method taking two `atomic` parameters while Listing 7.14 presents the result of applying the parameter transformation. Each atomic parameter is transformed individually and any `ref` or `out` modifiers are preserved.

Listing 7.13: `atomic` Parameters Before Transformation

```
public class AtomicParameterExample
{
  public void TestMethod(atomic int x, bool b, atomic ref string
      s)
  {
    //Body
  }
}
```

Listing 7.14: `atomic` Parameters After Transformation

```
public class AtomicParameterExample
{
  public void TestMethod(TMInt x, bool b, ref TMVar<string> s)
  {
    //Body
  }
}
```

7.3.6.1 Transactional Output Parameters

Transactional output parameters described in Section 5.2.3 require additional handling as C# requires every execution path in the method body to assign a value to the parameter[52, p. 42]. As described in Section 7.3.5, any assignment to a variable is replaced with an assignment to its `Value` property. As a consequence no assignments occur to the transactional parameter itself, which results in an error in the generated code. To rectify this error, an assignment, assigning a new STM object with the same type as the parameter, is generated in the top of the method body for every `atomic out` parameter of that particular method. For each `atomic out` parameter the AC# transformation contains the following steps:

1. Construct an object initialization expression that creates a new STM object of the same type as the parameter.
2. Construct an assignment statement that assigns the newly constructed object initialization expression to the `atomic out` parameter.

3. Insert the assignment statement as the first statement in the body of the enclosing method declaration.

Listing 7.15 shows a method with an `atomic out` parameter while Listing 7.16 shows the result of applying the transformation. Based on the `atomic out` parameter declared on line 3 of Listing 7.15 the assignment on line 5 of Listing 7.16 is generated.

Listing 7.15: `atomic out` Parameter Before Transformation

```
1 public class AtomicOutExample
2 {
3   public static void TestMethodAtomic(atomic out int i, int j)
4   {
5     i = 12;
6     j = 12;
7   }
8 }
```

Listing 7.16: `atomic out` Parameter After Transformation

```
1 public class AtomicOutExample
2 {
3   public static void TestMethodAtomic(out TMInt i, int j)
4   {
5     i = new TMInt();
6     i.Value = 12;
7     j = 12;
8   }
9 }
```

While the generated assignment solves the problem of assignments to `atomic out` parameters it introduces two new problems. Firstly, C# requires that every `out` parameter is assigned a value before a method exits, generating a compile time error if that is not the case[17, p. 94]. This error message is lost for transactional output parameters due to the generated assignment. Secondly, C# requires that an `out` parameter is assigned to before reading from it[17, p. 94], generating a compile time error if a read occurs before an `out` parameter is assigned a value. If a read occurs before any assignment in the original source code, no error is given due to the generated assignment. In both cases a meaningful error can be generated by applying analysis to a transactional output parameters `Value` property by following the rules defined in[17, p. 95]. No such analysis has however been implemented in the initial prototype of AC#.

7.3.7 Arguments

As described in Section 5.2.1 AC# support value parameters. Replacing the type of an atomic parameter with the corresponding STM type, as described in Section 7.3.6, results in a type mismatch when calling a method with an `atomic` parameter of some type T, as the parameter is no longer of type T, but of T's corresponding STM type. As a result transformation must be applied to arguments passed to an `atomic` parameter, ensuring that the argument represents an object of the required STM type. The transformation applied when an argument is passed to an `atomic` parameter goes through the following steps:

1. Determine the STM type of the formal parameter to which the argument corresponds.
2. Construct an object initialization expression, which creates an object of the previously determined STM type, where the argument expression is given as argument to the constructor of the object.
3. Replace the argument with the constructed object initialization expression.

Listing 7.17 presents an example of calling a method with an `atomic` parameter, while Listing 7.18 shows the result of the transformation. Line 3 is transformed as described in Section 7.3.6. Line 10 in Listing 7.17 is transformed to line 10 in Listing 7.18. The argument to the `atomic` parameter is replaced with an object initialization expression that creates a new `TMInt` object which is initialized using the original argument.

Listing 7.17: `atomic` Argument Before Transformation

```
 1 public class AtomicArgumentExample
 2 {
 3   public static void TestMethod(atomic int x, int y)
 4   {
 5     //Body
 6   }
 7
 8   public static void Main()
 9   {
10     TestMethod(1, 2);
11   }
12 }
```

Listing 7.18: `atomic` Argument After Transformation

```
public class AtomicArgumentExample
{
  public static void TestMethod(TMInt x, int y)
  {
    //Body
  }

  public static void Main()
  {
    TestMethod(new TMInt(1), 2);
  }
}
```

7.3.7.1 Ref/Out Arguments

Supporting `atomic` `ref` and `out` parameters, as described in Section 5.2.4.1, presents a problem due to the type mismatch as a result of transforming `atomic` variables. `ref` and `out` parameters require the argument to be an assignable variable of the same type as the parameter. However as the type of the parameter is transformed, a variable of type T cannot be passed directly as `ref` or `out` to an `atomic` parameter of type T.

Four different cases exist for passing `ref` and `out` arguments. These are:

1. $T \rightarrow$ `atomic` T
2. `atomic` $T \rightarrow T$
3. `atomic` $T \rightarrow$ `atomic` T
4. $T \rightarrow T$

where T is some arbitrary type, *atomic* T is T's corresponding STM type and *atomic* $T \rightarrow T$ represents passing an argument of type T into a parameter of the STM type corresponding to T. The third and fourth cases are handled by the C# compiler so transformation must only be applied in the first and second cases. For each argument node in the syntax tree the transformation goes through the following steps:

1. Determine whether the argument represents one of the two previously described cases. If so, the remaining transformation steps are applied.
2. Construct a local variable with a type definition equal to that of the parameter, initialized using the original argument, and insert it just before the method call.

3. Replace the original argument with an identifier equal to the name of the previously generated intermediate local variable, passed using the same modifier as the original argument

4. Construct an assignment syntax node that assigns the value of the intermediate local variable to the original argument and insert it just after the method call

Listing 7.19 shows an example containing three method calls, where the first and third call falls in one of the categories which require transformation. Applying the transformation to the example presented in Listing 7.19 produces a syntax tree representing the code shown in Listing 7.20. Line 18 of Listing 7.19 is transformed to lines 18 to 20 of Listing 7.20. As the parameter is `atomic`, the local variable inserted on line 18 is of the parameters corresponding STM type. The argument to the method call has been replaced as seen on line 19. The assignment on line 20 assigns the value computed by the called method to the original argument. Line 19 of Listing 7.19 is not transformed as it represents the case `atomic` $T \rightarrow$ `atomic` T. Line 20 of Listing 7.19 is transformed much like 18, except the generated local field is of type `int` instead of an STM type, as the parameter is not declared `atomic`.

C# requires the argument for a `ref` or `out` parameter to be a variable. For the case $T \rightarrow$ `atomic` T and `atomic` $T \rightarrow T$, compile time analysis has been implemented to generate an error if the original argument does not correspond to a variable, as this is always the case in the generated code.

Listing 7.19: `ref` Arguments Before Transformation

```
1  public class AtomicRefExample
2  {
3    public static void TestMethodAtomic(atomic ref int i)
4    {
5      i = 12;
6    }
7
8    public static void TestMethod(ref int i)
9    {
10     i = 12;
11   }
12
13   public static void Main()
14   {
15     int i = 10;
16     atomic int iAtomic = 10;
17
18     TestMethodAtomic(ref i);
19     TestMethodAtomic(ref iAtomic);
20     TestMethod(ref iAtomic);
21   }
22 }
```

Listing 7.20: `ref` Arguments After Transformation

```
public class AtomicRefExample
{
  public static void TestMethodAtomic(ref TMInt i)
  {
    i.Value = 12;
  }

  public static void TestMethod(ref int i)
  {
    i = 12;
  }

  public static void Main()
  {
    int i = 10;
    TMInt iAtomic = new TMInt(10);

    TMInt _gen1 = new TMInt(i);
    TestMethodAtomic(ref _gen1);
    i = _gen1.Value;

    TestMethodAtomic(ref iAtomic);

    int _gen3 = iAtomic.Value;
    TestMethod(ref _gen3);
    iAtomic.Value = _gen3;
  }
}
```

7.3.8 Retry

In Section 5.4 the design of conditional synchronization through the use of `retry` is described. As `retry` is a keyword used as a statement, much like the `break` and `continue` statements in C#, the transformation need to identify all `retry` keywords and replace them with a method invocation to the static method `STMSystem.Retry` defined in the STM library, described in Section 6.3.3. The library then carries out the effect of the `retry` statement. To inform programmers of unintended behavior, analysis generating an error if the retry statement is used outside of an `atomic` or `orelse` block, has been implemented.

7.4 Testing

Testing is employed to ensure the Roslyn extension fulfills the design decisions of AC# and the integration with existing language features described in Chapter 5.

For the reasons described in Section 6.5, unit testing is chosen for testing the Roslyn extension. Unit testing is valuable in relation to the maintenance and further development of the Roslyn extension. Furthermore, Roslyn is still under heavy development, which means radical changes will happen. The unit test suite is beneficial to test if the extension still works, whenever new changes are made to Roslyn. The disadvantage of unit testing being more time consuming than ad hoc testing is outweighed by the many advantages which is gained. Additionally, each unit test is built using a black-box testing strategy[69, p. 87], where the program is viewed as a black box and focus is only on the input and output. This is done as the focus of the unit tests is on the functionality of the compiler. That is, the tests focus on what it does, and not on how it does it. Additionally it fits well with how a compiler is normally treated, as it is treated as a black-box where the input and output is only available.

Each STM construct and each integration with an existing language feature is covered by at least a single unit test. Furthermore, integration with disallowed existing language features is tested, e.g. there is a unit test that ensures an error is produced if the programmer tries to use the `retry` statement outside of an `atomic` or `orelse` block. This results in a rather extensive test suite, which is valuable in relation to the correctness of the STM constructs and integration with language features of AC#.

Listing 7.21 shows an example of a unit test for an empty atomic block. On line 3 the string `finalStr` to be compiled is generated using the `MakeSkeletonWithMain` method, which generates a test namespace, class and main method, with the supplied string arguments `strInClass` and `strInMain` included. Afterwards the string is written to a test file, which is used as argument in the method on line 8, which executes `csc.exe` (C# command line compiler) and returns a `CmdRes` object that contains the command line output. On line 9 it is checked that the command line output does not contain any warning, error or invalid exitcode. For unit tests that check error generations, e.g. an illegal existing language feature used with an STM construct, this step is not present, instead `CmdRes` can be checked if it contains the expected error or warning. On line 11 the expected result of the compilation string `expecStr` is generated and on line 17 the actual compilation string `compiledStr` is fetched. In order to retrieve the actual compilation string, csc.exe is extended with an additional argument called `stmiout`, which writes the source code after STM transformation to the specified path. This argument is used in the method on line 8. Finally on line 18 the expected string and the compiled string is checked for equality, if equal the test passes and otherwise it fails. Before checking for equality, the `AssertEqualStrings` method removes formatting from the strings.

Listing 7.21: Empty Atomic Block Unit Test

```
public void AtomicBlockEmpty()
{
    string finalStr = MakeSkeletonWithMain(
      strInClass: "",
      strInMain: "atomic{\n\t\t\t"+
        "}");
    StringToTestFile(finalStr);
    CmdRes res = RunCscWithOutAndStmIOut();
    AssertCmdRes(res);

    string expecStr = MakeSkeletonWithMain(
      strInClass: "",
      strInMain: STM.STMNameSpace + ".STMSystem.Atomic(" +
        "() =>" +
        "{" +
        "});");
    string compiledStr = TestFileToString(currentCompiledCsFile);
    AssertEqualStrings(expecStr, compiledStr);
}
```

7.5 Design and Integration Revisited

This section describes areas in which the initial prototype implementation described in this chapter, conflicts with the intended design and language integration described in Chapter 5. Ideally no conflicts would happen, as the analysis in the design and language integration chapter would foresee them. However, implementing the design gave an insight and knowledge into unforeseen areas.

7.5.1 Transactional Local Variables and Fields

As described in Section 7.3.2 and Section 5.1.4 local variables and fields of type T declared `atomic` are transformed to declarations with a type equal to T's corresponding STM type. As part of this process the local variable or field is initialized to an STM object of the correct type, using any initializer expression if present. If not, the value of the variable is initialized to the default value of type T through the STM object's parameter less constructor.

For fields this presents no issue as fields are always initialized to the default value of their type if no initializer expression is present[17, p. 93]. This is however not the case for local variables. The C# compiler generates an error if a local variable is accessed before it has been assigned a value[17, p. 96]. As `atomic` local variables are always initialized, this error will never occur for such variables, which can lead to unintended behavior in cases where the programmer forgets to assign an `atomic` local variable. In such cases the default value of the `atomic` local variables original type will be the value accessed, potentially leading to some confusion.

In order to provide similar error messages for `atomic` local variables and regular C# local variables, the Roslyn flow analysis which detects whether a given accessed local variable has been definitely assigned, must be extended. It must be extended to track both initial assignment, as part of the variable declaration, and assignment to the `STM` objects `Value` property, as opposed to assignments directly to the variable. Such analysis has however not been included in the initial prototype described in this chapter.

7.5.2 Transactional Optional Parameters

C# requires that the default value given when declaring an optional parameter is one of the following[17, p. 309]:

1. A constant expression
2. An expression of the form `new S()` where `S` is a value type
3. An expression of the form `default(S)` where `S` is a value type

As described in Section 7.3.6 the transformation of an `atomic` parameter transforms the type of the parameter to one of the STM types. Providing a default value to an `atomic` parameter therefore amounts to initializing a new STM object with the same type as the parameter, supplied with the defined default value to its constructor. All STM types are however reference types and the creation of a new STM object can therefore, per the three rules described above, not be supplied as the replacement default value, as this would result in a compile time error. As a result default values for transactional value parameters are not implemented in the initial prototype described in this chapter. Analysis has been implemented to produce an error if an `atomic` value parameter is given a default value. This is done in the parsing phase, by checking if a parameter is declared with the `atomic` modifier and has an optional value.

7.5.3 Transactional Ref/Out Parameters & Arguments

As described in Section 5.2.2 and Section 5.2.3 the intended design was that assignments to transactional `ref/out` parameters would take effect when the transaction in which the assignment takes place commits. This behavior holds true for the case:

- $atomic\ \ T \quad \rightarrow \quad atomic\ \ T$

That is, the case involving an `atomic` argument and parameter. For the cases:

- $T \quad \rightarrow \quad atomic \;\; T$
- $atomic \;\; T \quad \rightarrow \quad T$

the behavior is however different. Due to the type mismatch between the argument and parameter, an intermediate local variable is passed as the actual argument, and the original argument is assigned as the first thing after the call finishes. This conflicts with traditional C#, where any assignment to a `ref/out` parameter takes effect immediately[52, p. 76]. If the parameter is declared `atomic` and the method call is wrapped in an `atomic` block, there will be no difference in timing as any assignments to atomic variables take effect when the transaction commits. However, if the call is not wrapped in an `atomic` block the difference in timing can have unintended consequences, especially when considering concurrent execution. Due to this issue a number of ways for handling `atomic ref/out` parameters and arguments were investigated:

1. Warn the programmer if one of the two problematic cases is used.
2. Disallow the two problematic cases described above.
3. Disallow `atomic ref/out` parameters.

In C#, passing a volatile field as `ref/out` results in the field being treated as non-volatile within the method body[70]. If such an operation is detected the compiler creates a warning informing the programmer of the change in semantics. The same approach could be taken whenever one of the two problematic cases described earlier is detected.

Disallowing the `ref/out` cases where both an `atomic` and non-`atomic` variable/parameter are involved removes the cases where the desired timing cannot be provided. However, it will, for example, no longer be possible to pass an `atomic` integer into an `out` integer parameter such as that of `Int32.TryParse`. Such restrictions limits the flexibility when using `atomic` variables, and the programmer needs to circumvent this, by the use of intermediate variables which clutters the code, and reduces the readability.

Disallowing `atomic ref/out` parameters, prevents the programmer from declaring `atomic ref/out` parameters, which removes the timing issues present in the cases where an intermediate `atomic` variable is required. However, the ability to pass an `atomic` variable by reference is lost and the case of passing a variable of type `atomic` T into a parameter of type T is still unsolved. Consequently, this would limit the orthogonality of the `atomic` construct.

For the initial prototype presented in this chapter the first option of warning the programmer of the change in semantics has been selected. This ensures that the change in semantics is not hidden from the programmer thereby allowing her to adapt the program. This choice follows the approach taken by C# in the case of a `volatile` field passed by ref into a method as described above.

8 Evaluation of Characteristics

This chapter evaluates AC#, its associated STM library, and locking in C# according to the evaluation method described in Section 1.5. The evaluation facilitates a conclusion on our hypothesis *"Language integrated STM provides a valid alternative to locking in terms of usability, and provides additional benefits compared to library-based STM, when solving concurrent problems in C#"* defined in Section 1.4. The following sections describes how each of the three concurrency approaches are evaluated according to the characteristics. Each of the three concurrency approaches is given a placement on the spectrum of a given characteristic, as defined in Section 1.5.

8.1 Implicit or Explicit Concurrency

All the selected concurrency approaches rely on starting threads in order to introduce concurrency as well as manually specifying critical regions using either locks or the `atomic` block. This makes all the approaches lean toward the explicit end of the spectrum. The two STM based approaches are however more implicit as STM abstracts away synchronization details. Locking in C# on the other hand requires explicitly stating how synchronization is achieved and is therefore placed close to the explicit concurrency extreme. AC# and the STM library reside slightly more towards the implicit concurrency end of the spectrum. The placement of the three concurrency approaches on the implicit - explicit concurrency spectrum is depicted in Figure 8.1.

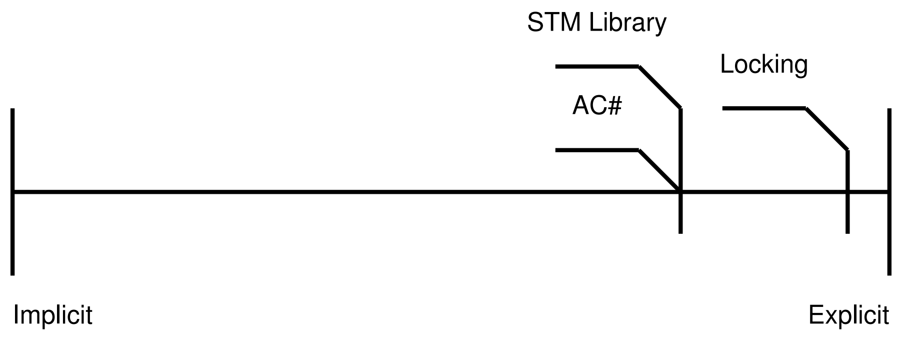


Figure 8.1: Concurrency approaches on the implicit - explicit concurrency spectrum

8.2 Fault Restrictive or Expressive

Locking presents the programmer with a set of constructs aimed at solving concurrency problems, but does little to guarantee correct usage. The locking constructs in C# are explained in detail in Section 2.1. Locking enables control of synchronization at a very low level of detail, which is very expressive. Therefore, locking in C# is placed at the expressive end of the spectrum.

The STM based approaches delegate the details of how synchronization is achieved to the underlying STM system, allowing STM based concurrency to avoid some of the errors associated with locking, such as deadlocks. The STM based approaches however still rely on shared memory for communication and require programmers to define transaction scopes and introduce concurrency by starting threads. The abstractions provided by STM limits the possibility of expressing synchronization at a low level of detail. The STM based approaches reside towards the expressive end of the spectrum but contain fault restrictive elements pulling them more towards the fault restrictive extreme than locking. AC# however implements a number of compile time errors and warnings keeping the programmer from utilizing undesirable combinations such as `retry` statements outside `atomic` or `orelse` block, moving it slightly towards the fault-restrictive end of the spectrum. The placement of the three concurrency approaches on the fault restrictive - expressive spectrum is depicted in Figure 8.2.

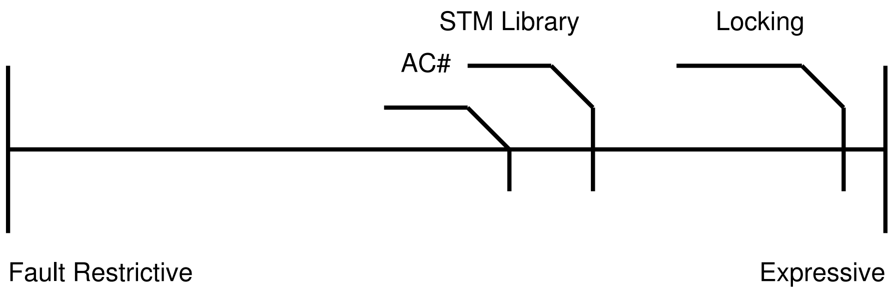


Figure 8.2: Concurrency approaches on the fault restrictive - expressive spectrum

8.3 Pessimistic or Optimistic

Locking applies synchronization by enforcing mutual exclusion. This approach is well suited in scenarios where errors are common which would result in a high number of aborts if an STM based approach had been used. Locking is therefore an inherently pessimistic concurrency approach as it prevents errors from occurring instead of correcting them when they occur. STM on the other hand allows multiple threads to proceed simultaneously, correcting any errors that may occur by aborting and re-executing transactions. Hence STM takes an optimistic approach to concurrency. However, the employed STM system uses lazy updates as opposed to eager updates, which is a slightly more pessimistic approach to STM keeping AC# and the STM library from reaching the optimistic extreme.Therefore AC# and the STM library resides close to at the optimistic extreme of the pessimistic - optimistic spectrum while locking resides close to at the pessimistic extreme. The placement of the three concurrency approaches on the pessimistic - optimistic spectrum is depicted in Figure 8.3.

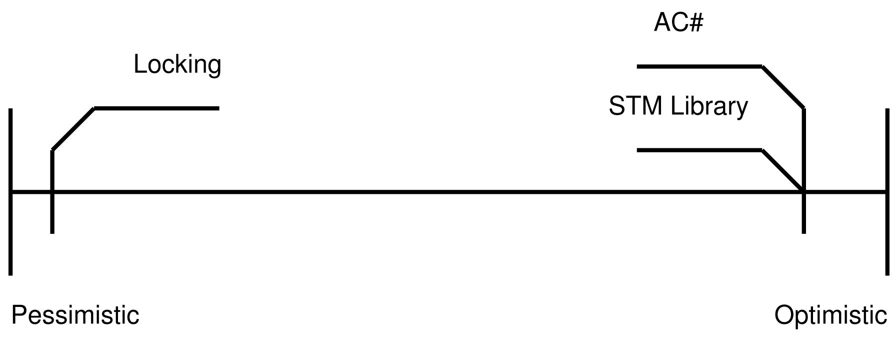


Figure 8.3: Concurrency approaches on the pessimistic - optimistic spectrum

8.4 Readability & Writability

The evaluation of readability and writability is based on a number of shared characteristics: Data Types, Syntax Design, Simplicity, and Orthogonality. The characteristics Data Types and Syntax Design will not be evaluated on a spectrum of two extremes, as the choices made are trade-offs affecting other characteristics, e.g. simplicity and readability. The evaluation of these will therefore be taken into account when the other characteristics are evaluated.

In addition to the shared characteristics, writability is based on level of abstraction and expressivity.

8.4.1 Data Types

All three concurrency approaches are either integrated into or build around C#, which means they have almost the same data types. A difference lies in the STM approaches where the types of transactional variables are treated differently. In the STM library, transactional variables are defined using transactional types, such as `TMInt`, which means that it is possible to create an array or list containing these types, e.g. `TMInt[]` defines an array of `TMInt` types. AC# instead preserves the original types of transactional variables, allowing the programmer to utilize an `atomic` variable of type T as if it was of type T even though the compiler transforms the type of the variable to T's corresponding STM type, as described in Section 7.3.2. As transactional variables are not special types in AC# it is not possible to define a list or array of transactional variables directly, instead a wrapper class with an `atomic` field must be employed.

The need to define an array of transactional types was experienced during the development of the hashmap implementations which use an array to represent the buckets of the hashmap. Listing 8.1 and Listing 8.2 show how the hashmap buckets are defined in the STM library and AC#, respectively. In the library STM example on line 3 the buckets are defined directly. The type of the `_buckets` field will be explained gradually. The inner `Node` type, is a simple class which represents a key value pair, where the value is a transactional variable, defined in the STM library code on line 8 and in AC# code on line 12. In case of collisions the `Node` class allows for chaining of nodes representing the key/value pairs inserted into the hashmap through its `Next` property. Each bucket is represented by a transactional variable which points to the first element in the collision list, if any such element exists. The `Next` property of the `Node` class is transactional, allowing the system to detect changes to the internals of the collision list. This, along with the transactional variable pointing to the first element in the list, allows for changes to the entire collision list to be tracked. Finally the backing array is also wrapped in a transactional variable allowing the STM system to track assignments to the entire backing array in order track when the backing array is resized. In the AC# example on line 3 it is not possible to define the array of transactional variables directly, so a wrapper type called `Bucket` defined on line 7 is used. This is a consequence of the design of the AC#.

Ultimately prohibiting the programmer to define an array or list of transactional types directly, lowers the readability and writability of AC#, as it requires the programmer to write, read and maintain an extra wrapper class. To remedy this, a library of data structures which provides tracking to internal changes could be included in the language.

The special types used by the STM library to represent transactional variables requires the programmer to use the `Value` property whenever she needs to make an assignment. Implicit conversion ensures that an object of one of the STM types, such as `TMInt`, in most cases can be used as an object of the type it is wrapping. Listing 8.3 shows an equality comparison of two transactional variables using the STM library. The comparison on line 5 requires the programmer to access the `Value` property of the `TMVar` objects as implicit conversion will not be used if the `TMVar`'s are compared directly. If the value property had not been used the `TMVar` objects would have been compared, producing an incorrect result. A similar problem exists when calling

Listing 8.1: HashMap Buckets Array - STM Library

```
public class StmHashMap<K,V> : BaseHashMap<K,V>
{
  private readonly TMVar<TMVar<Node>[]> _buckets =
  new TMVar<TMVar<Node>[]>();

  ...  //Other code

  private class Node
  {
    public K Key { get; private set; }
    public TMVar<V> Value { get; private set; }
    public TMVar<Node> Next { get; private set; }
    public Node(K key, V value)
    {
      Key = key;
      Value = new TMVar<V>(value);
      Next = new TMVar<Node>();
    }
  }
}
```

Listing 8.2: HashMap Buckets Array - AC#

```
public class StmHashMap<K,V> : BaseHashMap<K,V>
{
  private atomic Bucket[] _buckets;

  ... //Other code

  private class Bucket
  {
    public atomic Node Value { get; set; }
  }

  private class Node
  {
    public K Key { get; private set; }
    public atomic V Value { get; set; }
    public atomic Node Next { get; set; }
    public Node(K key, V value)
    {
      Key = key;
      Value = value;
    }
  }
}
```

Listing 8.3: Equality comparison of `TMVar<bool>`

```
public bool TestMethod()
{
  TMVar<bool> v1 = new TMVar<bool>(false);
  TMVar<bool> v2 = new TMVar<bool>(false);
  return v1.Value == v2.Value;
}
```

a method on a `TMVar` object, implicit conversion does not allow methods of the wrapped type to be called on the `TMVar` directly.

The two STM based approaches have problems combining with existing data structures. For example, if a programmer utilizes a `List<T>` inside a transaction, any operations on the list are not tracked by the STM system as `List<T>` is not implemented using transactional variables. Therefore operations on the list take effect immediately. A transaction may be executed multiple times due to being aborted and retried, resulting in the operations, such as adding an element to the list, being executed and taking effect multiple times. Consequently data structures utilized in transactions must be: *a*) Implemented using transactional variables, allowing the STM system to track any changes, or *b*) Immutable and tracked through assignment to a transactional variable.

8.4.2 Syntax Design

The use of keywords reduce simplicity but increase readability, as the intent stands out clearly[21, p. 12-13]. Locking in C# employs only a single keyword, `lock`. The rest of the functionality is provided by library calls as described in Section 2.1. The use of libraries for locking constructs makes it blend in with other library code, even though it has a special purpose related to synchronization. This decreases the readability, but keeps the simplicity of the language as it does not introduce keywords for all constructs.

The STM library also uses library calls as described above, thus it has the same disadvantages and the readability is decreased compared to AC# which has keywords for the special constructs. The four keywords introduced in AC# do however decrease the simplicity of the language. As the `atomic` keyword has different meanings depending on the context in AC#, its readability is reduced. The simplicity is however raised by employing a minimum of keywords.

The STM library requires the programmer to use static method calls, wrap transactional code in lambdas, wrap transactional variables in `TMVars`, use the `Value` property when getting or setting a value, supply `orelse` transactions as arguments to the atomic method call, and write return in front of the static method call if a method must return the value computed by a transaction. All of these concerns are abstracted away in AC#.

In C# it is possible to name variables after reserved keywords, by prefixing the name with `@`. Due to this, the impact of the additional keywords in AC# on the naming of variables is reduced and it is distinguishable when `atomic` is used as a name or a keyword.

8.4.3 Low or High Simplicity

Locking is based on the idea of mutual exclusion. C# supplies a number of different constructs for defining synchronization using locking as described in Section 2.1. Applying locking correctly in complex scenarios is considered to be hard[2, p. 56], mainly due to the number of errors that can arise, such as deadlocks.

Locking requires the programmer to explicitly state how synchronization is to be applied. Both library-based STM and AC# take a more declarative approach than locking when specifying synchronization, thus it is simpler as the programmer does not have to worry about low level details. The usability studies produced by Rossbach et al. in [42] and Pankratius et al. in [43] find that the surveyed students found STM simpler than fine grained locking but more complex than coarse grained locking due to issues understanding the execution of transactions.

In case of conditional synchronization, both forms of STM can leverage the `retry` functionality, making a transaction block until the variables previously read by the transaction are changed. In C# a `Monitor` allows blocking until another thread notifies the blocked thread, but contrary to `retry` this is not on a declarative level. The `retry` statement was used in the two STM based queue implementations to, with only two lines of code, make the calling thread block until the queue is non-empty in case `Dequeue` is called on an empty queue. The same feature in the lock-based implementation would require the use of a semaphore or the `Monitor` class's `Wait` and `Pulse/PulseAll` methods, which both require the programmer to manage low level details.

Listing 8.4 and Listing 8.5 show the `Add` method for the locking and library-based STM hashmap implementations respectively. The locking hashmap implementation divides the backing array into a number of stripes, where each stripe is protected by its own lock. When accessing a bucket the hashmap must determine what lock to acquire based on the index of the bucket. This strategy is further described in Appendix C.1.4. In Listing 8.4 on line 4, a calculation is made to determine which lock needs to be used, additionally on line 18 another lock is needed for synchronizing access to the `_size` variable as the previously acquired lock only covers part of the buckets, meaning that another thread could be changing the `_size` variable. This is avoided in Listing 8.5 where the method is wrapped in an `atomic` block, and the STM system detects and resolves potential conflicts.

Library-based STM does not have support for checking implicit dependency between static calls at compile time, e.g. a `retry` outside an `atomic` call will not produce a compile time error. Additionally, the STM types provided must be used to track variables in transactions. These types cause a type mismatch when used together with standard types, thus making the library more complex to use. This issue is partly resolved by providing implicit type conversion on the STM types, but the implicit conversion is not possible in all cases. In the cases where it is not, the programmer must access the wrapped value by the `Value` property available on all STM types.

AC# introduces additional language constructs which reduces the simplicity of the language. It does however increase the simplicity of using the STM part, e.g. using `retry` outside of an `atomic` block will result in a compile time error. Additionally, `atomic` can also be used along with fields, local variables, and parameters, making them traceable in transactions without having to use specific STM types. This simplifies interaction between code that uses `atomic` variables and code that does not.

Listing 8.4: ConcurrentHashMap `Add` Method - Locking

```
public override void Add(K key, V value)
{
    var hashCode = GetHashCode(key);
    lock (_locks[GetLockIndex(hashCode)])
    {
        var bucket = _buckets[GetBucketIndex(hashCode)];
        var node = FindNode(bucket, key);

        if (node != null)
        {
            //If node is not null, key exist in map. Update the
               value
            node.Value = value;
        }
        else
        {
            //Else insert the node
            bucket.AddFirst(CreateNode(key, value));
            lock (_sizeLock)
            {
                _size++;
            }
        }
    }

    ResizeIfNeeded();
}
```

Listing 8.5: ConcurrentHashMap `Add` Method - AC#

```
public override void Add(K key, V value)
{
    atomic
    {
        var bucketIndex = GetBucketIndex(key);
        //TMVar wrapping the immutable chain list
        var bucketVar = _buckets[bucketIndex];
        var node = FindNode(key, bucketVar.Value);

        if (node != null)
        {
            //If node is not null key exist in map. Update the
               value
            node.Value = value;
        }
        else
        {
            //Else insert the node
            bucketVar.Value = bucketVar.Value.Add(CreateNode(key,
               value));
            _size++;
            ResizeIfNeeded();
        }
    }
}
```

The placement of the three concurrency approaches on the low simplicity - high simplicity spectrum is depicted in Figure 8.4.

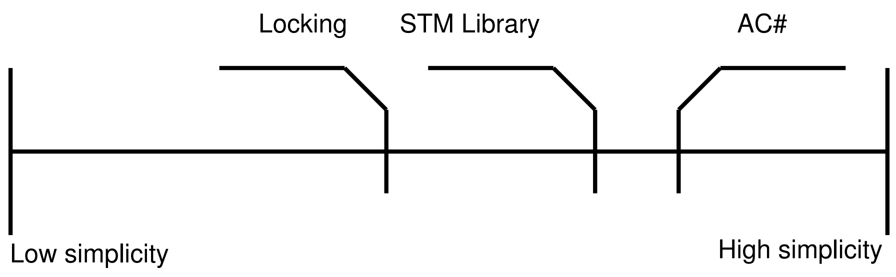


Figure 8.4: Concurrency approaches on the low - high simplicity spectrum

8.4.4 Low or High Orthogonality

As described in Section 2.1, locking encompasses a number of basic constructs aiding in different concurrency scenarios. These constructs can be combined to handle complex concurrency issues. Some of these combinations are however erroneous, producing hard to debug problems such as deadlocks. Locking does not put any restraints on the language features with which it can be combined and may therefore seem to be highly orthogonal. The risk of deadlocks when combining lock-based implementations or employing multiple locks limit the orthogonality, keeping locking from reaching the high orthogonality extreme. Locking in C# provides no solutions to these issues. As a result locking in C# is placed between the middle and the high orthogonality extreme.

STM removes the issue of deadlocks and allows STM based code segments to be combined using transactional nesting. STM combines poorly with irreversible actions, that is actions which cannot be rolled back in case a transaction aborts such as IO. Neither the STM library nor AC# offers a solution to this problem significantly reducing their orthogonality. Furthermore both the STM library and AC# have problems combining with existing data structures reducing their orthogonality further.

As described in Section 8.4.1 the programmer cannot rely on implicit conversion from an STM object to the value it wraps in all cases, reducing the orthogonality of the STM library. The STM library however benefits from the advantages provided by STM, but is hampered by the not being able to combine with irreversible actions and existing data structures. Consequently library-based STM is placed just above the middle on the low - high orthogonality spectrum. AC# treats transactional variables as an object of their original type, removing the type mismatch between the regular types and the corresponding STM types. As a result, AC# allows transactional variables to be compared directly and methods can be invoked directly on the transactional variable, without having to reason about implicit conversion or manually accessing the `Value` property, as described in Section 8.4.1. Therefore AC# is placed further towards the high orthogonality end of the spectrum than library-based STM. AC# however remain more towards the low orthogonality extreme than locking.

The placement of the three concurrency approaches on the low - high orthogonality spectrum is depicted in Figure 8.5.

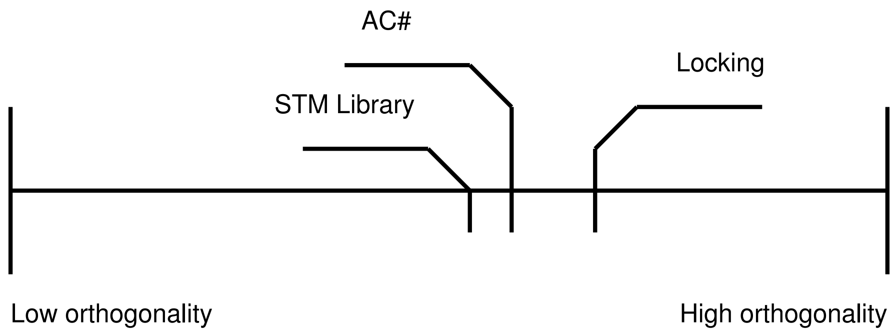


Figure 8.5: concurrency approaches on the low - high orthogonality spectrum

8.4.5 Low or High Readability

As described in Section 8.4.2, locking in C#, except the lock statement, is provided by library calls, and library STM is only provided by library calls. This negatively affect their readability because these calls blends in with other library code. To avoid this confusion AC# uses special keywords which positively affects its readability, however it also negatively affects simplicity and thereby the readability because the programmer has to know these keyword. Also the `atomic` keyword has different meanings depending on the context. This negatively affects the readability, as the programmer must be aware of these different meanings. Much of the boilerplate code necessary in the STM library is abstracted away in AC#, e.g. use of the Value property, which improves its readability as there is less code to reason about.

In Section 8.4.1 the data types characteristic is described. The STM library treats transactional variables as special types, where AC# treats transactional variables as regular types with an associated atomic keyword modifier. As a result it is not possible to define an array or list of transactional types directly in AC#, which reduces its readability, because of the extra boilerplate code that the required wrapper class introduces. Additionally the STM library allows for implicit conversion when reading a transactional value, thereby avoiding the specification of the Value property, which positively affects the readability. Implicit conversion can however produce incorrect results and is not applicable in all scenarios, so ultimately the readability suffers.

In Section 8.4.3 the simplicity of locking is placed close to the low simplicity extreme, where the STM library is placed a bit higher than the middle of the spectrum and AC# even higher. In Section 8.4.4 the orthogonality of locking resides between the middle and high end of the spectrum, while library-based STM is placed just above the middle of the spectrum with AC# just above it. These characteristics directly influence the readability of the concurrency approaches.

All the concurrency approaches have to explicitly mark critical regions of code in order to ensure a program is race condition free. This negatively affects their readability, as by reading the program it is hard to know whether critical regions are marked correctly everywhere, and it is especially hard in large programs.

The basic `lock` construct and idea behind locking is simple, if a resource is locked, only a single thread is allowed access to the resource. This positively affects the readability of locking and in small code segments it may seem highly readable. However locking suffers in particular from the concurrency issue of deadlocks. This negatively affects the readability of locking drastically, as in order to reason about deadlocks, the programmer has to reason about every code segment where locking is applied and how these segments interact. As locking code can be fragmented it can become hard for the programmer to reason about. Additionally, if a program makes use of other libraries that uses locking internally, the programmer also have to reason about the locks contained within those libraries, as locks do not compose. The readability of locking is therefore placed towards low readability on the spectrum.

The STM approaches remove the issue of deadlocks, which positively affects the readability, as the programmer does not have to reason about deadlocks. Additionally, STM also removes the issue of composing synchronized code segments, as STM allows nested transactions. Similar to locking, STM suffers from code fragmentation which can make it hard to get an overview of all the STM synchronization in a program, which affects the readability negatively. Furthermore understanding the concept of memory transactions can, as described in [42], at first be hard to grasp, as it promotes a new way of synchronizing programs. This problem is worsened since both the `orelse` and `retry` constructs have been included in the STM library and AC#. Additionally, AC# is able to produce compile time errors and warnings, e.g. when a `retry` keyword is defined outside an `atomic` scope, which improves its readability over the STM library, which does not have this capability. Library STM have a type mismatch between STM types and regular types, which AC# does not. Therefore AC# has higher readability than the STM library.

Ultimately the STM library is placed close to the middle on the spectrum and AC# further towards the high readability extreme. The main drawback for both approaches, is that the programmer has to mark critical regions of code which can be very fragmented. Additionally understanding the concept of memory transactions may at first be hard, especially as both the `orelse` and `retry` constructs have been introduced.

The placement of the three concurrency approaches on the low readability - high readability spectrum is shown in Figure 8.6.

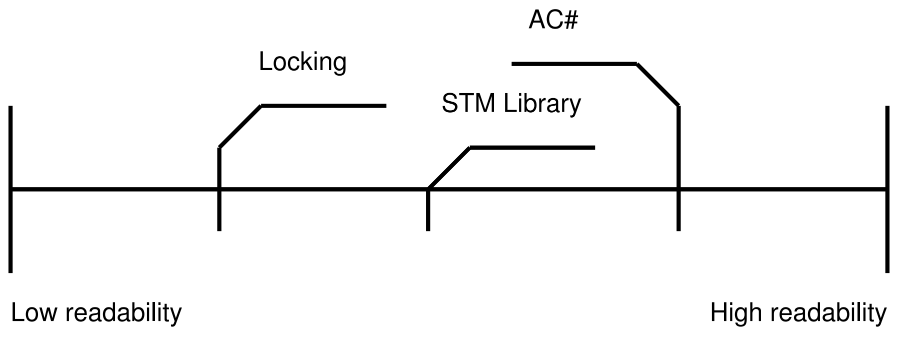


Figure 8.6: Concurrency approaches on the low - high readability spectrum

8.4.6 Low or High Level of Abstraction

Locking is tightly coupled with the hardware architecture through hardware instructions such as Test-and-set and Compare-and-swap[71, p. 1990]. Furthermore the low-level abstraction of threads isused in order to introduce concurrency. Additionally the programmer has to state exactly how synchronization should be applied using locks. C# offers the `lock` statement which provides a small abstraction over the release of a lock. The `lock` statement is however not applicable in all cases. If for example a timeout on the acquisition of a lock is required or a more advanced form of lock, such as a semaphore, is required, the `lock` keyword is not applicable. That is the case in the locking Dining Philosophers implementation, which requires the second lock to be taken with a timeout. Furthermore the locking Santa Claus implementation uses semaphores to signal between Santa, the elfs and the reindeer. The `lock` keyword does not provide such capabilities. Additionally correct usage of the `Monitor` class leads to the use of a `try/finally` block to ensure that the acquired lock is released in case of an exception[72], reducing the level of abstraction. Ultimately locking in C# is placed close to the low level of abstraction extreme of the spectrum with the small abstractions provided keeping it from being at the extreme.

The STM approaches also rely on threads, but provide a higher level of abstraction for synchronization. STM uses transactions, where code segments are marked and the details of how synchronization is achieved are abstracted away by the STM system. Both AC# and the STM library facilities strong atomicity, as described in Section 4.3 and the STM system will therefore manage both transactional and non-transactional access, thus keeping the level of abstraction high. Based on the above, the general STM approach lies between the middle and high end of the spectrum, where the main drawbacks is that the programmer still has to manage threads and mark regions of code that should be synchronized.

Some differences in the level of abstraction exist with regards to AC# and the STM library. As described in Section 8.4.2 the STM library requires the programmer to use static method calls and lambdas for defining transactions. AC# substitutes this with language based support for the `atomic` block. Listing 8.6 and Listing 8.7 present an implementation of the `SleepUntilAwoken` method from the library and AC# Santa Claus problem implementations, respectively. The method ensures that Santa sleeps until he is awoken by either three elfs at his door, or all reindeers back from vacation, ready to fly his sleigh. The `return` statement on line 3 of Listing 8.6 is not present in the AC# implementation as it is abstracted away by AC#. Furthermore the static method call and creation of lambas to represent transaction bodies on line 3 is abstracted away by the `atomic` block and orelse block on lines 3 and 11 of Listing 8.7.

Listing 8.6: `SleepUntilAwoken` Method - STM Library

```
private WakeState SleepUntilAwoken()
{
  return STMSystem.Atomic(() =>
  {
    if (_rBuffer.Count != SCStats.NR_REINDEER)
    {
      STMSystem.Retry();
    }
    return WakeState.ReindeerBack;
  },
    () =>
    {
      if (_eBuffer.Count != SCStats.MAX_ELFS)
      {
        STMSystem.Retry();
      }
      return WakeState.ElfsIncompetent;
    });
}
```

Listing 8.7: `SleepUntilAwoken` Method - STM Language

```
private WakeState SleepUntilAwoken()
{
  atomic
  {
    if (_rBuffer.Count != SantaClausProblem.NR_REINDEER)
    {
      retry;
    }
    return WakeState.ReindeerBack;
  }
  orelse
  {
    if (_eBuffer.Count != SantaClausProblem.MAX_ELFS)
    {
      retry;
    }
    return WakeState.ElfsIncompetent;
  }
}
```

The STM library exposes transactional variables as special STM types, where AC# instead allows variables to be marked as `atomic`, letting the programmer preserve the original type. AC# is therefore considered to have a higher level of abstraction than library STM. As described previously STM however combines poorly with irreversible actions leaving it up to the programmer to correctly handle actions such as IO in combination with transactions. As neither the STM library nor AC# can abstract over these problem, their level of abstraction is reduced. Therefore library-based STM is placed just above the middle of the spectrum while AC# resides further towards the high level of abstraction extreme.

The placement of the three concurrency approaches on the low level of abstraction - high level of abstraction spectrum is depicted in Figure 8.7.

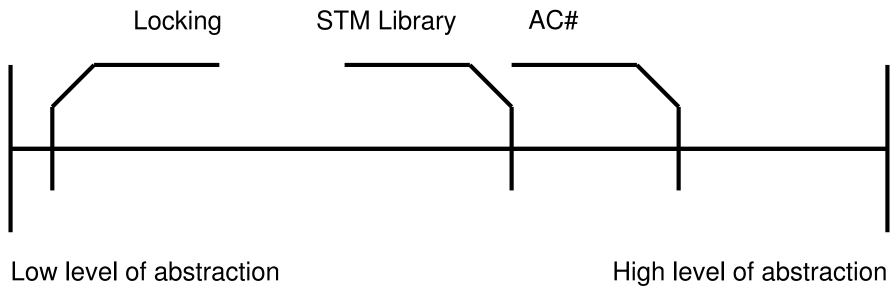


Figure 8.7: Concurrency approaches on the low - high level of abstraction spectrum

8.4.7 Low or High Expressivity

The locking constructs in C# present different ways to express exactly how synchronization should be applied, thereby affecting the expressivity positively. Since locking is prone to a number of concurrency related issues, the expressivity is severely reduced as the programmers focus is limited by the complex synchronization form. Locking in C# gives the programmer control over many low level details concerning how synchronization is applied but at the same time requires the programmer to specify these details. This along with the risk of deadlocks and other issues limits how the programmer can express functionality concisely and conveniently. Locking is therefore placed towards low expressivity, being drawn a bit towards the middle of the spectrum because of the many choices in locking constructs C# offers.

STM also builds upon threads which limits its expressivity. Memory transactions however represent a more declarative and expressive approach than locking, as the programmer only has to specify critical regions, allowing the STM system to manage the details of how synchronization is applied. This affects the expressivity positively, as it accomplishes a great deal of computation with little code. Furthermore STM eliminates the issues of deadlocks which makes it more convenient to express synchronization, as the programmer does not have to reason about it. However, neither in the STM library nor AC# a convenient way to express irreversible actions exists, such as exceptions and IO, the programmer is left alone in ensuring these actions work correctly with transactions, which negativity affects the expressivity. Furthermore, having to rely solely on transactional or immutable data structures limits the expressivity of the STM based approaches. Based on the above, the general STM approach lies between the middle and high end of the spectrum.

Some differences with regards to expressivity between library-based STM and AC# exist. The expressivity of the approaches is closely related to their level of abstraction, described in Section 8.4.6. As described in Section 8.4.2, library STM requires the programmer to put extra effort into expressing the intended functionality e.g. wrapping transactional code in lambdas, wrapping transactional variables in `TMVar` types and using the `Value` property when getting or setting a value on a transactional variable. That is abstracted away in AC# which makes it more expressive as it provides a more convenient and less tedious way of specifying computations. However, the abstraction that AC# provides, disallows the programmer from defining an array of transactional types directly and instead requires the programmer to use a wrapper class, as described in Section 8.4.1. Based on the above, AC# is considered to have a higher expressivity than library STM because of the higher level of abstraction that it facilitates. The difference is however limited as AC# does not allow for the definition of an array or list of transactional types as described above.

The placement of the three concurrency approaches on the low expressivity - high expressivity spectrum is depicted in Figure 8.8.

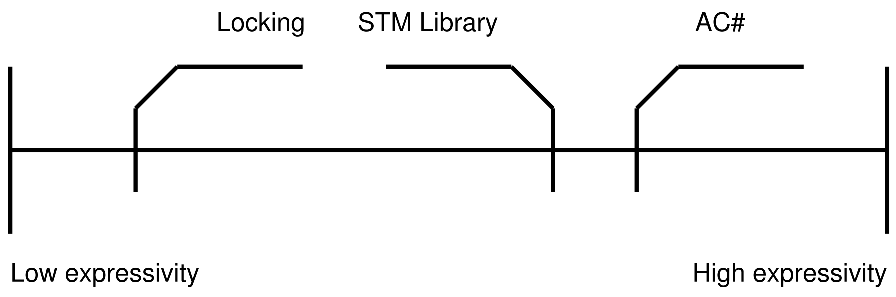


Figure 8.8: Concurrency approaches on the low - high expressivity spectrum

8.4.8 Low or High Writability

As discussed in Section 8.4.6 locking in C# has a low level of abstraction due to its usage of low level constructs. The `lock` statement provides an abstraction over the acquisition and release of a lock, providing increased writability in cases where it is applicable. The writability of locking in C# is however reduced as a result of the low level of abstraction. The STM library's level of abstraction is placed just above the middle towards the high level of abstraction end of the spectrum, while AC# has a higher level of abstraction than the STM library. This positively impacts the writability of the STM library and AC#.

The locking constructs described in Section 2.1 are prone to common locking problems, such as deadlocks, resulting in a low simplicity score which negatively impacts its writability. The expressivity of the STM library resides above the middle of the spectrum, reduced by the need to wrap transactional code in lambdas, wrap transactional variables in `TMVar` types and use the `Value` property when getting or setting a value on a transactional variable, hindering the programmer from concisely expressing the intended behavior, resulting in reduced writability. AC# removes this burden from the programmer, by delegating the work to the compiler, resulting in a higher expressivity score and improved writability.

With respect to simplicity, locking in C# is, as described in Section 8.4.3, placed close to the low simplicity end of the spectrum mainly due the number of errors that can arise. This negatively impacts the approach's writability. The two STM based approaches benefit from a declarative approach to defining synchronization. The STM library is placed just above the middle of the spectrum, while AC# is placed towards the high simplicity end of the spectrum. AC# is given a higher simplicity score due to its cleaner syntax and the ability to use existing types instead of the STM equivalents, resulting in a greater positive impact on writability.

As described in Section 8.4.4 locking in C# has orthogonal properties but it is limited due to the risk of errors when combining locking constructs. The STM based approaches benefit from simplified composition based on transactional nesting but simultaneously introduces issues with irreversible actions such as IO as well as existing data structures. The STM library has problems combining with existing types in all cases. Furthermore the implicit conversion feature cannot be relied on in all cases, requiring the programmer to access the `Value` property, reducing the orthogonality of the STM library. Ultimately this results in the STM library being placed just above the middle on the low - high orthogonality spectrum. AC# addresses many of the issues present in the `STM` library by allowing the programmer to use any existing types instead of the STM types and by delegating much of the work required by the STM library to the compiler. This results in AC# being placed just above the STM library on the low - high orthogonality spectrum. Locking is however placed further towards the high orthogonality extreme due to its ability to correctly combine with most language constructs, excluding locking itself.

Locking suffers from the risk of a number of serious errors, resulting in it being hard to use in complex scenarios, limiting its writability. The STM based approaches solve many of the errors present with locking but introduces new problems with irreversible actions and data structures. The STM library has problems with types, accessing the wrapped value, and the syntax of transaction definitions, which limits its writability. AC# addresses many of the writability problems present in the STM library by delegating the work to the compiler. Both STM based approaches however require that data types, including collections, are constructed using transactional variables or implemented as immutable in order to interact with the STM system, reducing their writability. Based on these observations and the evaluations of the previous characteristics, the three approaches have been placed on the low - high writability spectrum as depicted in Figure 8.9.

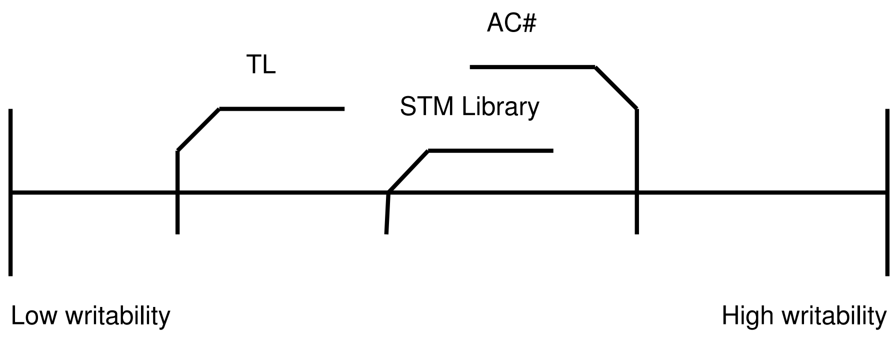


Figure 8.9: Concurrency approaches on the low - high writability spectrum

9 Conclusion

This thesis tests the hypothesis: *"Language integrated STM provides a valid alternative to locking in terms of usability, and provides additional benefits compared to library-based STM, when solving concurrent problems in C#"*. In order to do so an extension to C#, called AC# was designed and implemented. AC# provides language integrated support for STM which allows programmers to utilize transactions alongside existing C# features. The STM system powering AC# was implemented as a C# library. To implement AC#, the Roslyn C# compiler was extended to transform AC# source code into regular C# code, which utilizes the aforementioned STM library to execute any defined transactions. For each of the concurrency approaches: AC#, the STM library and locking in C#, implementations of the Dining Philosophers problem, the Santa Claus problem, a concurrent queue, and a concurrent hashmap were created. These implementations served as the basis for a usability evaluation according to an extended version of the characteristics defined in our previous study[3].

In order to develop AC# a set of requirements were defined, detailing how the underlying STM system should behave in relation to for example tracking granularity, atomicity level and nesting. Based on the requirements AC# was designed. The design includes a description of new language constructs as well as a description of modifications to existing language features. A number of STM implementations were investigated. Based on this investigation as well as the requirements and design, an STM library, utilizing the TLII algorithm, was implemented. The STM library was tested using a number of unit tests, ensuring that transactions are executed correctly. In order to perform the actual integration of STM into C#, the open source Roslyn C# compiler was extended. This required a deep knowledge of the Roslyn project and its structure, which initiated an investigation of Roslyn. Due to the limited availability of literature with regards to Roslyn, much of the knowledge obtained in this area was acquired by reading and debugging Roslyn's source code. A number of unit tests were defined for the Roslyn extension, in order to ensure that all STM constructs and the integration with the existing language features work correctly. Finally AC#, the STM library and locking in C# were evaluated according to a number of usability related characteristics, facilitating a conclusion upon the hypothesis.

This chapter addresses the problem statement questions and summarizes the evaluation of the characteristics in order to conclude on our hypothesis.

9.1 Problem Statement Questions Revisited

The goal of this master thesis was to investigate the usability of language integrated support for STM in C#, compared to a library-based solution and existing locking features. This section will address the problem statement questions used for structuring our investigation.

Selecting features to an STM system for C# was done by analyzing different approaches to tracking granularity, transactions & variables, atomicity, side-effects, conditional synchronization, nesting, and opacity. These features determined the requirements for the design. The requirements and the existing language features of C# were taken into account in the design and integration process of AC#, identifying how to make an integration encompassing: atomic blocks, transactional variables, parameters & arguments, conditional synchronization, and nesting. The implementation strategies McRT, TLII, and JVSTM were analyzed based on the criteria from the requirements. TLII was selected based on its fit with strong atomicity and retry, as well as being well documented. To find out how the Roslyn compiler was structured, a review of whitepapers, blogs, forum posts, and the source code were conducted. This resulted in a description of Roslyns compiler phases, syntax tree, and API. This knowledge allowed us to extend Roslyn to encompass the features of AC#, by modifying its lexer, parser, syntax tree and symbols. Additionally, the Roslyn compiler was extended with a transformation phase, transforming AC#'s constructs to ordinary C# code, allowing the remaining compilation phases to be reused.

9.2 Hypothesis Revisited

With a functioning AC# language, library-based STM, and locking in C#, we evaluated the concurrency approaches based on cases representing different aspects of concurrency. Based on this evaluation, we conclude on the hypothesis in Section 9.2.1.

STM has a more implicit degree of concurrency than locking, as synchronization details are abstracted away. This also makes library-based STM more fault restrictive than locking as synchronization details are handled correctly by the underlying STM system, hindering potential errors known from locking e.g. deadlocks. AC# is even more fault restrictive, since static analysis identifies a number of errors. Both locking and STM use shared memory for communication, and require the programmer to specify the scope of synchronization reducing their readability and writability.

The optimistic nature of STM relies on conflicts being uncommon, correcting them if they occur. The correction results in aborting a transaction, and running it again. Thus all effects occurring in transactions must be able to be discarded in case of an abort. This is however not the case for irreversible actions, decreasing the usability of STM, as the programmer is required to know which operations cannot be used in transactions.

From the programmer's point of view AC# does not change the type of transactional variables, which increases its readability. This is a benefit compared to the library-based STM approach, which requires transactional variables to be wrapped in `TMVar` objects. The programmer can for example compare two transactional variables directly in AC#, where in the STM library she is required to compare the `Value` properties of the `TMVars` instead. Furthermore AC# and the STM library require data types, including collections, to be build using transactional variables which lowers the writability of the two STM based approaches.

AC# allows the programmer to define transactional variables by supplying the `atomic` modifier. As a consequence an array of transactional variables cannot be defined in AC#. Instead the programmer must either wrap each element in an object with a transactional field, employ an immutable list, or use a specialized STM list that tracks elements internally. These workarounds do however reduce both readability and writability, which negatively affects the usability of AC#.

AC# integrates STM at a level similar to that of locking in C#, providing a number of benefits compared to library-based STM. The STM related keywords supplied by AC# increases its readability as the intent stands out clearly. However, simplicity is reduced, as special keywords add complexity to the language. The syntax of AC# is improved compared to library-based STM, as boilerplate syntax is abstracted away. This is a benefit to AC# compared to library-based STM.

The declarative synchronization approach in STM provides high simplicity, a high level of abstraction and high expressivity. Locking requires the programmer to explicitly control synchronization details, which is hard to get right. STM on the other hand handles synchronization as long as the critical regions are specified, which improves usability. Additionally AC# provides static analysis, capable of identifying errors in the code, a feature not present in the STM library.

The locking constructs provided by C# can be combined to handle different scenarios which improves the orthogonality. However the risk of deadlocks when combining locks reduces the orthogonality. STM removes the issue of deadlocks, and allows transactions to be nested, which is highly orthogonal. A major caveat is, that STM cannot be combined with irreversible actions, keeping the orthogonality, level of abstraction and expressivity of the STM based approaches from reaching their potential, impacting both readability and writability. Library-based STM supplies implicit conversion from an STM object to its wrapped value. However certain cases, e.g. comparisons, require explicitly accessing the `Value` property of the STM object. AC# does not have the type mismatch between regular types and the corresponding STM types. Due to this, no extra actions must be taken when retrieving the value of a transactional variable. This provides a benefit to AC# over library-based STM, as it improves writability.

9.2.1 Final Conclusion

Based on the above findings we conclude that language integrated STM provides a valid alternative to locking when solving concurrency problems. This is based on it being evaluated as having more implicit concurrency, being more fault restrictive, being simpler, having a level of abstraction, expressivity, readability, and writability. Locking does however have higher orthogonality due to STM's problems combining with irreversible actions and existing data structures, reducing the applicability of AC#.

Furthermore, AC# provides additional benefits over language integrated STM. This is based on being evaluated as equal in implicit concurrency and optimism, while having improved readability and writability based on a less intrusive syntax design, higher simplicity, orthogonality, level of abstraction, and expressivity.

10 Reflection

This chapter presents a reflection on the decisions made throughout the report. Section 10.1 presents a reflection on the findings of the preliminary analysis. Section 10.2 describes reflections related to transactional local variables and parameters as well as `ref/out` arguments and parameters. Reflections related to the STM library, described in Chapter 7, are presented in Section 10.3. Finally, the reflections related to the Roslyn extension and C# 6.0 are covered in Section 10.4 and Section 10.5 respectively.

10.1 Preliminary Investigation

A preliminary investigation was conducted, in order to reduce the risk associated with the Roslyn compiler and the implementation of an STM system.

A number of papers describing STM system implementations were investigated. This gave us a deeper understanding of the known approaches for implementing STM systems. To get some hands on experience, a prototype STM system based on the two implementations described in [53, p. 424] was developed. Developing such a prototype system furthered our understanding of the different implementation strategies.

As the Roslyn compiler was released 10 months prior to the start of the project, the amount of literature on the subject is limited, consisting of mostly: a white paper[30], documentation associated with the Roslyn Github repository[73], sample implementations and walkthroughs[74], as well as blog posts. Most of these sources describe the compiler API as opposed to the structure of the compiler. To further our understanding of the Roslyn compiler we have investigated these sources. In addition, an exploratory modification of the compiler was done to investigate its structure. The lexer and parser of the compiler were modified to handle the syntax of an `atomic` block. The exploratory implementation furthered our knowledge of the compilers structure and API design, and served as an onset into the deeper exploration of the compiler documented in Chapter 3.

Getting preliminary hands on experience with the required technologies at an early stage of the project, helped ensure the feasibility for the implementations and provided the project group with confidence in the feasibility of the project. Bringing more information to the table at an early stage assisted in deciding on the implementation strategies for the STM system and for the extension of the Roslyn compiler.

10.2 Design

These sections describes the project group's reflections over areas related to the design of AC# described in Chapter 5.

10.2.1 Transactional Local Variables & Parameters

In order to improve the orthogonality and flexibility of AC# the design allows for the declaration of transactional local variables and parameters. Neither of these features have however been used in the implementations on which the evaluation is build, indicating that the features may not be required. The reason may be that the problems selected in this thesis, described in Section 1.5.1, does not fit well with the use of transactional local variables and parameters, where other types of problems and applications may fit better. It may also be that the way we have implemented the selected problems did not present a reason to use transactional local variables and parameters, where other ways may have.

The initial idea behind allowing transactional local variables was that local variables can be captured in a lambda expression which is executed by multiple threads, resulting in the need for synchronization. The option to do so was however not employed in the evaluation implementations. Allowing the programmer to declare `atomic` local variables instead of relying solely on `atomic` fields makes AC# more orthogonal but also less simple.

A parameter is similar to a local variable which is initialized using its corresponding argument[52, p. 76]. Therefore the idea behind allowing transactional parameters is similar to that of transactional local variables. If transactional parameters were not allowed, the programmer would have to declare an `atomic` local variable and initialize it to the value of the parameter, if she wished to utilize the parameter as a variable in a transaction. As with transactional local variables, transactional parameters improve the orthogonality of AC# but reduce the simplicity.

C# does not allow for the declaration of `volatile` local variables and parameters, but only `volatile` fields. Similarly allowing only `atomic` fields may be sufficient for AC#. Determining whether these features are truly required, through further studies e.g. usability studies, could assist in furthering AC# as well as aid others seeking to supply language integrated STM.

10.2.2 Transactional Ref/Out Parameters & Arguments

The initial idea behind including transactional ref/out parameters was to preserve the functionality provided by C# with regards to transactional arguments and parameters, thereby providing an integration with existing language

Listing 10.1: STM library interfaced based on applying functions

```
public class Account
{
  private double _balance;

  public void AtomicTransfer(Account other, double amount)
  {
    STMSystem.Atomic(() =>
    {
      var newBalance = STMSystem.TMRead(ref _balance)+amount;
      STMSystem.TMWrite(ref _balance, newBalance);
      newBalance = STMSystem.TMRead(ref other._balance)-amount;
      STMSystem.TMWrite(ref other._balance, newBalance);
    });
  }
}
```

features. As with transactional local variables and parameters, transactional ref/out parameters were however not used in the implementations for the evaluation, suggesting that further work is required to determine whether the feature is required or if it simply complicates AC#, reducing its simplicity.

As described in Section 7.5.3, providing the intended semantics for transactional ref/out parameters & arguments proved problematic due to the types of transactional variables being substituted with their corresponding STM types. As a result the simplicity of AC# is reduced due to the programmer having to reason about the changed semantics.

To remove the problem, a library with an interface based on central metadata and applying functions to variables in order to read/modify could be adopted. Such an interface will remove the need for transforming the types during compilation, allowing transactional variables to be passed directly into non-transactional parameters and vice versa. If a transactional variable is passed by ref/out into a method, transactional access will however still be lost, as the body of the method does not treat the variable as transactional. As described in Section 7.5.3 C# utilizes a similar approach for `volatile` variables passed by ref/out.

Listing 10.1 presents an example of how an `STM` interface based on applying functions to variables could be designed. The `STMSystem.TMRead` and `STMSystem.TMWrite` are applied to the `_balance` variables in order to read and write their associated value.

10.3 STM Implementation

This section describes the project groups reflections related to the implementation of the STM system described in Chapter 6.

10.3.1 STM Algorithm

The implemented STM system utilizes the TLII algorithm[12]. TLII uses commit time locking to ensure that any values written by a transaction becomes visible to the remaining system as a single atomic step. The TLII algorithm is well documented [12][53, p. 438][9, p. 106], which allowed us to gain deep insight into the workings of the algorithm, easing the implementation of more advanced features such as orelse, retry and nesting, which are not described in the reference materials.

Due to the utilization of locking the TLII algorithm is unable to supply any of the progress guarantees described in Section 6.1.4. Selecting an algorithm which could supply one of these progress guarantees would have positively affected the simplicity of the STM library and of AC#. In case of Wait-freedom the programmer would know that all transactions would eventually commit allowing her to utilize this knowledge in the implementation. Alternatively the inclusion of a contention manager, dictating conflict resolution, into the existing implementation could be investigated. A contention manager may allow the existing implementation to provide similar guarantees in the absence of failures.

10.3.2 STM Library Interface

As described in Section 1.5, locking in C#, library based STM and AC# is evaluated according to a set of characteristics. As the STM library was evaluated according to its usability, its interface was required to be designed to maximize usability, while still satisfying the defined requirements. AC# however does not impose such a requirement on its backing STM library. The programmer does not see the transformed code, removing the need for an STM library with high usability. In fact it may be advantageous for performance to utilize a backing STM library with a high degree of flexibility, allowing the compiler to optimize the generated code. As described in Section 10.2.2 adopting a library interface based on central metadata and applying functions to a variable in order to read and modify, will remove the issues related to the types of transactional variables and `ref`/`out` parameters.

10.4 Roslyn Extension

As described in Section 7.1 the Roslyn compiler was extended at the level of the syntax tree/symbol information. The lexer, parser, syntax tree, and symbol table were modified to support the constructs described in Chapter 5, and transformations were applied to the modified syntax tree, producing a standard C# syntax tree that represents the execution of transactions as calls to the STM library. The transformed syntax tree is passed to the remaining compiler phases, utilizing the existing semantic analysis and code generation of the Roslyn compiler.

Alternatively the transformation could be applied at the level of what in Roslyn is referred to as the bound tree. The bound tree is a semantic representation of the source code in the form of a tree structure. The bound tree is constructed based on the syntax tree and is the basis for the code generation phase. During the construction of the bound tree, semantic analysis and source code transformation is applied. As described in Section 7.1 applying transformations to the syntax tree causes it to lose its roundtripable property, due to the syntax tree no longer representing the original AC# source code. The transformations required for executing transactions could be applied along with the standard C# transformations, as part of constructing the bound tree. This would cause the syntax tree to preserve its roundtripable property but will however require the implementation of the transformation to be mingled with the existing C# transformation implementations and semantic analysis, making the implementations more complex. Alternatively the transformations could be applied after the bound tree has been constructed. However this approach requires that the logic which constructs the bound tree, as well as the semantic analysis, is modified to support the STM constructs described in Chapter 5. The bound tree and the associated transformation and analysis implementations are not part of the public API provided by the Roslyn compiler due to it often undergoing significant changes[37]. Furthermore any documentation is limited to blog/forum post as well as the comments embedded in the source code. Due to these factors, applying transformations at the level of the bound tree is significantly more complex than the approach utilized in Section 7.1. Furthermore, it may require the transformation implementation to be rewritten if AC# is moved to a newer version of C# and the bound tree has undergone significant changes since the previously utilized implementation was created.

10.5 C# 6.0

As described in Section 1.3, the development of AC# is based on C# 5, which at the time of writing is the most recent version of C#. C# 6.0 is, as of May 2015, however under active development and a number of new features have been shown to the public. This section presents our reflections upon how the new features impact the findings of this project. The new features supplied by C# 6.0 are listed in [75] and described in [76], [77] and [78]. Only new features relevant to locking in C#, AC# or the STM library are described.

Listing 10.2: STM library with static using statement

```
private WakeState SleepUntilAwoken()
{
  return Atomic(() =>
  {
    if (_rBuffer.Count != SCStats.NR_REINDEER) {
      Retry();
    }
    return WakeState.ReindeerBack;
  },
    () =>
    {
      if (_eBuffer.Count != SCStats.MAX_ELFS) {
        Retry();
      }
      return WakeState.ElfsIncompetent;
    });
}
```

10.5.1 Static Using Statements

Static using statements allows the programmer to import the static methods of a particular class, giving her the ability to call these methods as if they were methods of the class in which the call takes place[76][78]. This feature is of interest to programmers utilizing the STM library as it will allow them to call the `atomic` and `retry` methods of the `STMSystem` class without having to specify the class for each call, thereby improving the writability of the library. Listing 10.2 shows how the `SleepUntilAwoken` method of the STM library based Santa Claus problem implementation could look if the static methods of the `STMSystem` class were imported through a static using statement.

10.5.2 Auto-Property Initializers

Auto-Property intializers allow the programmer to supply an initializer expression to the definition of an Auto-Property[76][77][78]. This causes the property to be initialized using the initializer expression. If the Auto-Property defines only a getter, the backing field is automatically declared readonly[76][78].

The Roslyn extension will have to be modified to handle `atomic` Auto-properties with an initializer expression. The extension must transform an initializer expression to an object creation expression which creates a new STM object of the STM type corresponding to the type of the `atomic` property, and initialize it using the defined initializer expression. Instead of initializing the property directly, the initializer should be applied to the backing field generated as part of the transformation of `atomic` auto-properties described in Section 7.3.3.

11 Future Work

This chapter presents possibilities for extending upon the work done in this thesis. Section 11.1 suggests a performance test in order to examine how AC# performs in comparison to locking in C#. Section 11.2 suggests a deep STM integration in order to achieve better performance. Finally, Section 11.3 suggests to investigate possible ways of combining irreversible actions and STM in the context AC#.

11.1 Performance Test

The focus of this thesis has been on evaluating the usability of language integrated STM compared to that of library-based STM and locking. As described in Section 1.3, the performance of STM has not yet been considered. However having language integrated STM is not sufficient, as concurrency usually is introduced to achieve good performance. It is essential to know how AC# performs in comparison to locking in C#, since if it performs far worse, it is not a valid alternative, in its current state. A performance test is therefore a candidate for future work, and the outcome may result in changes to the design and underlying algorithm of AC#, in order to improve its performance.

To investigate performance, an extensive performance suite with a number of problems or algorithms with great diversity could provide valuable insights into how the three concurrency approaches perform under varied circumstances. In this thesis, four concurrency problems have been developed, which advantageously could be included in the test suite, by measuring their performance. Furthermore a number of concurrent performance suites already exists, which could be investigated in order to help select additional problems. In [79] a performance suite, specifically for the actor model is proposed. The performance suite contains 28 test cases that *"range from popular microbenchmarks to classical concurrency problems to applications that demonstrate various styles of parallelism"* which are *"diversive, realistic and compute intensive"*. The dining philosophers and concurrent hashmap[1] are also included in this performance suite. Another benchmark specifically designed for STM is proposed in [80], which uses *"a set of workloads that correspond to realistic, complex, object-oriented applications which benefit from multi-threading"* to compare the performance between STM implementations. Comparing AC# to other STM implementations, could give valuable insights to evaluate the relation between STM design decisions and performance.

[1]Refered to as Concurrent Dictionary in the article

11.2 Integration into CLR

Chapter 7 describes how STM is integrated into C# by modifying the Roslyn compiler to transform the source code containing STM language constructs to standard C# code which utilizes the STM library. As a result no modifications are required to the CLR as it receives byte code in the standard CIL format produced by the Roslyn C# compiler's code generator.

As an alternative, a deeper integration, similar to that of [11] and [14], could be utilized. One approach is to extend the CLR and its associated byte code format CIL with support for respectively, the execution and definition of transactions. Extending CIL with support for the definition of transactions allow the JIT compiler to apply further optimization to the execution of transactions. Additionally, if the executing hardware supports it, the CLR could delegate smaller transactions to the hardware providing a performance boost.

If CIL is extended with support for the definition of transactions, the Roslyn compiler will have to be extended as well, as the compiler must generate code utilizing the new CIL instructions. As a result the lexing, parsing, semantic analysis and code generation of the Roslyn compiler must be extended to support the STM constructs. However, the lexing and parsing implemented in this project can be reused for such an extension.

11.3 Irreversible Actions

As described in Section 1.3, solving the integration problems between transactions and irreversible actions, has not been an area of focus for this thesis. As irreversible actions in the context of STM still contains a number of issues, the area could benefit from an increased research effort. AC# offers no support of or warning against irreversible actions in transactions, which hurts its usability, as it leaves it up to the programmer to manually handle it correctly.

The integration between transactions and irreversible actions is therefore a candidate for future work, both in terms of how to manage it in AC#, but also in terms of STM in general. A number of possible integration approaches exist, we briefly scratched the surface in our prior study[3, p. 51-52]. One approach, presented in [11, p. 4], disallows the use of native calls inside transactions, by raising a runtime exception. A similar approach is, to enable the developer to mark a function, so the STM system is aware of its side effect. In Clojure this is possible with the #io macro. If a function is marked, and used in a transaction, a runtime exception will be raised. In [16] Harris et al. proposes another approach where IO libraries should implement an interface, allowing the STM system to initiate a callback when the transaction is committed, allowing the effect to be buffered until then. In [14] Duffy proposes using a well known strategy from the transaction theory[48], having the programmer supply on-commit and on-rollback actions to perform or compensate for the irreversible action. All of these approaches however either impose additional effort unto the programmer or completely disallow irreversible actions. A more elegant solution will benefit the area of STM in general and may instigate the adaptation of STM as a language feature.

Appendix

A Roslyn Compiler Call Chain

In Figure A.1 an overview of the compilation call chain is shown. A compilation starts with `csc`'s main method being invoked, which calls the static `Run` method on the `Csc` class. This method creates a new `compiler` object of type `Csc`. `Csc` is a subtype of `CSharpCompiler`, located in the `CSharp.CSharpCodeAnalysis.Desktop` project, which again is a subtype of `CommonCompiler`, located in `Core.CodeAnalysis`. This means that the creation of the `compiler` object, calls the constructors of `Csc`, `CSharpCompiler` and `CommonCompiler`. Afterwards the `Run` method is invoked on the `compiler` object, which further invokes its parent's `Run` method, which again invokes its parent's `Run` method. Finally the `RunCore` method in `CommonCompiler` is invoked, which contains the general code that controls the overall flow of the compiler pipeline, illustrated in Figure 3.1. The `RunCore` method will, for each phase, call the language specific implementation of the phase, located in either C# in `CSharp.CSharpCodeAnalysis.Portable` or VB in `VisualBasic.BasicCodeAnalysis.Portable`, through dynamic dispatch.

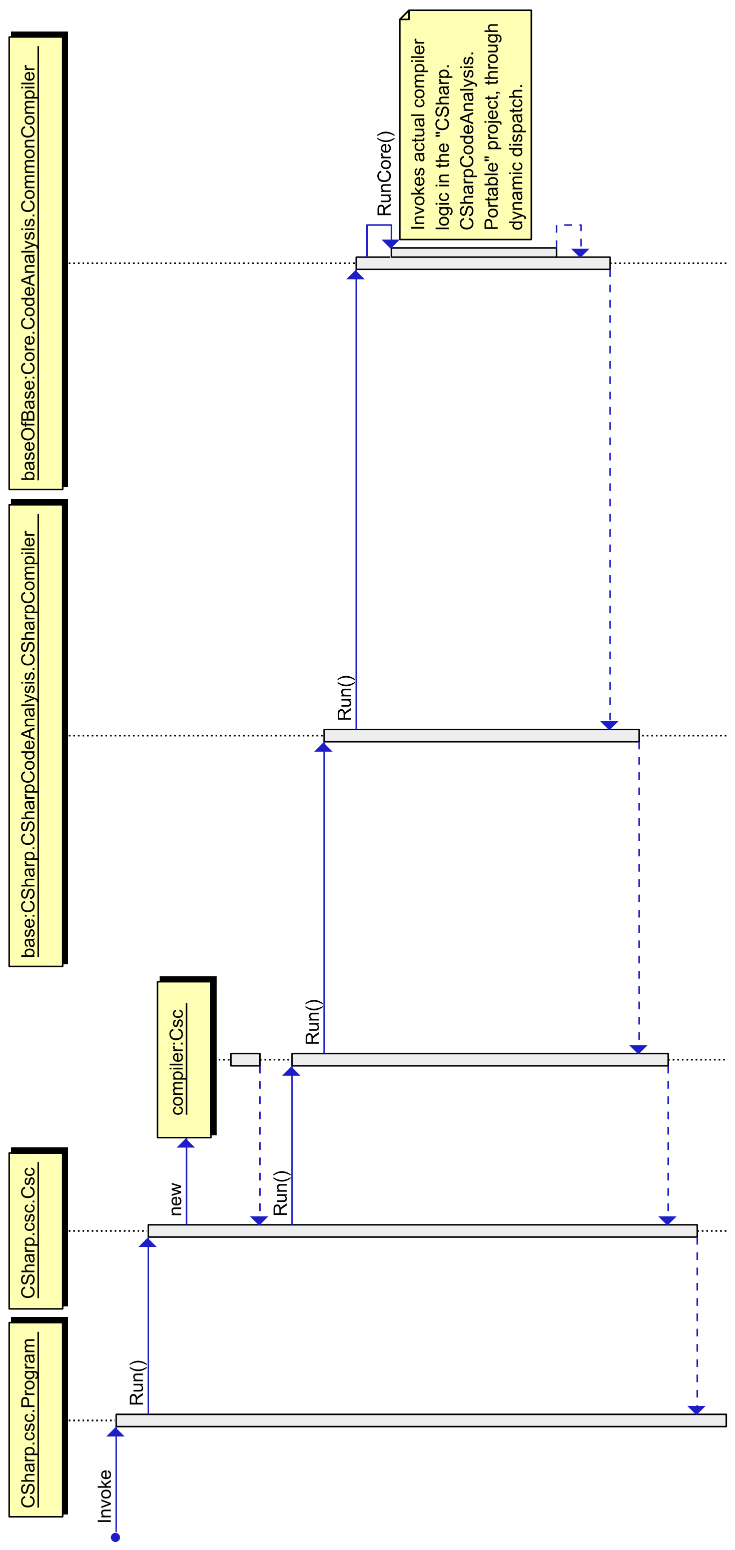

CSharp.csc.Program
CSharp.csc.Csc
compiler:Csc
base:CSharp.CSharpCodeAnalysis.CSharpCompiler
baseOfBase:Core.CodeAnalysis.CommonCompiler
Invoke
Run()
new
Run()
Run()
Run()
RunCore()
Invokes actual compiler logic in the "CSharp. CSharpCodeAnalysis. Portable" project, through dynamic dispatch.

Figure A.1: Sequence diagram showing an overview of the call chain of a C# compilation.

B Concurrency Problems

B.1 Dining Philosophers

The Dining Philosophers problem is defined as follows:

> *"Five philosophers spend their lives thinking and eating. The philosophers share a common dining room where there is a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a large bowl of spaghetti, and the table is laid with five forks (see Figure B.1). On feeling hungry, a philosopher enters the dining room, sits in his own chair, and picks up the fork on the left of his place. Unfortunately, the spaghetti is so tangled that he needs to pick up and use the fork on his right as well. When he has finished, he puts down both forks, and leaves the room. The room should keep a count of the number of philosophers in it."*
>
> *– Edgar Dijkstra[81, p. 673]*

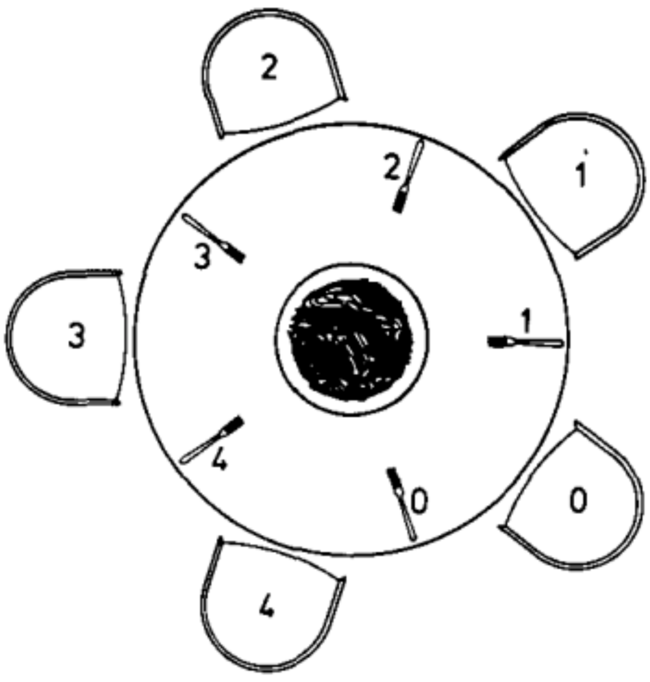


Figure B.1: The Dining Philosophers

B.2 The Santa Claus Problem

The Santa Claus problem is defined as follows:

> *"Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeer being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys. One elf's problem is never serious enough to wake up Santa (otherwise, he may never get any sleep), so, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer don't want to leave the tropics, and therefore they stay there until the last possible moment. They might not even come back, but since Santa is footing the bill for their year in paradise ... This could also explain the quickness in their delivering of presents, since the reindeer can't wait to get back to where it is warm.) The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh."*
>
> *– John A. Trono[82]*

B.3 Concurrent Queue

A queue is a First In First Out (FIFO) data structure, operating like the queue at a cash register[20, p. 234]. A queue supports the operations `Enqueue` and `Dequeue`. `Enqueue` inserts an item at the back of the queue while `Dequeue` gets and removes the item at the front of the queue, per the FIFO principle. A queue is commonly implemented using a linked list of nodes[20, p. 234]. To be able to enqueue and dequeue, the queue holds a reference to the first and last node in the list, referred to as the head and tail of the queue.

B.4 Concurrent HashMap

A HashMap is a data structure which allows looking up values based on their associated key and provides the operations Add, Get, and Remove. It benefits from the Get operation having an average time complexity of $O(1)$ under reasonable assumptions[20, p. 256]. Implementation details of a HashMap may vary, but traditionally they follow the concept described in[20, p. 256]. Internally it uses an array for storing a list of key/value pairs. When inserting a value into the HashMap, the array index is commonly calculated as:

$$Hash(key) \bmod arraylength \tag{B.1}$$

That is, a hash function applied to the key, modulo the size of the backing array. If another key is already stored in the array index calculated, it is said to be a collision. To handle this, the value is not stored directly in the array index, but in a list referred to as a bucket. This bucket is implemented as a linked list as this ensures fast insertions[20, p. 257]. Index based lookup and removal is not needed as the index of a particular key is not known. Therefore it is always required to iterate through the bucket list when searching for a particular key. The calculation of the bucket index must distribute the values evenly to lessen the collisions. When a certain percentage of the buckets are filled, the internal array must be resized to keep the risk of collisions low. This internal operation is expensive in terms of both time and space, as a new internal array of an increased size must be allocated, and the key/value pairs must be inserted into the new array.

C Evaluation Implementations

This appendix presents the implementations of the selected problems, described in Section 1.5, on which the evaluation presented in Chapter 8 is based. Along with each implementation follows a brief description.

C.1 Lock-Based

This sections presents the lock based implementations of the selected problems.

C.1.1 Dining Philosophers Problem

The lock based implementation of the dining philosophers problem requires the use of the `Monitor` class to acquire the second lock with a time out on line 35. As a result a `try/finally` block is used, on lines 33 to 48, to ensured that the acquired lock is released in case an error occurs while the lock is held.

Listing C.1: Lock Based Dining Philosophers Implementation

```
 1 public class LockingDiningPhilosophers
 2 {
 3   public static void Start()
 4   {
 5     var fork1 = new object();
 6     var fork2 = new object();
 7     var fork3 = new object();
 8     var fork4 = new object();
 9     var fork5 = new object();
10
11     var t1 = StartPhilosopher(fork1, fork2);
12     var t2 = StartPhilosopher(fork2, fork3);
13     var t3 = StartPhilosopher(fork3, fork4);
14     var t4 = StartPhilosopher(fork4, fork5);
15     var t5 = StartPhilosopher(fork5, fork1);
16
17     t1.Join();
18     t2.Join();
19     t3.Join();
20     t4.Join();
21     t5.Join();
22   }
23
24   private static Thread StartPhilosopher(object left, object
        right)
25   {
    var t1 = new Thread(() =>
    {
      while (true)
      {
        lock (left)
        {
          var lockTaken = false;
          try
          {
            Monitor.TryEnter(right, 100, ref lockTaken);
            if (lockTaken)
            {
              Console.WriteLine("Thread: " +
                  Thread.CurrentThread.ManagedThreadId + "
                  eating.");
              Thread.Sleep(100);
            }
          }
          finally
          {
            if (lockTaken)
            {
              Monitor.Exit(right);
            }
          }
        }
        Thread.Sleep(100);
      }
    });

    t1.Start();
    return t1;
  }
}
```

C.1.2 Santa Claus Problem

As seen on line 62, the lock Based Santa Claus problem implementation uses a semaphore to allow the elfs and reindeer to wake Santa, given their respective conditions are true. When Santa is awoken he must check whether he was awoken by the elfs or by the reindeer, as shown on lines 66 to 72. As the reindeer must take priority, as defined in Appendix B.2, their condition is checked in order to determine the action which Santa takes. Similarly, as shown on line 143, the elfs utilize a semaphore to ensure that only three elfs go to santa at a time. As shown on line 145 to 152 as well as 204 to 211, both the elfs and the reindeer enqueue themselves and check their condition while holding a lock on their respective queues, ensuring that only a single thread observes the condition as true and notifies Santa.

Listing C.2: Lock Based Santa Claus Implementation

```
public class LockingSantaClausProblem
{
  public static void Start()
  {
    var santaHandle = new SemaphoreSlim(0, 2);
    var sleigh = new SemaphoreSlim(0, SCStats.NR_REINDEER);
    var warmingHut = new SemaphoreSlim(0, SCStats.NR_REINDEER);
    var reindeerDone = new SemaphoreSlim(0, SCStats.NR_REINDEER);
    var elfWaiting = new SemaphoreSlim(0, SCStats.MAX_ELFS);
    var elfDone = new SemaphoreSlim(0, SCStats.MAX_ELFS);
    var maxElfs = new SemaphoreSlim(SCStats.MAX_ELFS,
        SCStats.MAX_ELFS);
    var rBuffer = new Queue<LockingReindeer>();
    var eBuffer = new Queue<LockingElf>();
    var santa = new LockingSanta(rBuffer, eBuffer, santaHandle,
        sleigh, warmingHut, reindeerDone, elfWaiting, elfDone);
    santa.Start();

    for (var i = 0; i < SCStats.NR_REINDEER; i++)
    {
      var reindeer = new LockingReindeer(i, rBuffer, santaHandle,
          sleigh, warmingHut, reindeerDone);
      reindeer.Start();
    }

    for (var i = 0; i < SCStats.NR_ELFS; i++)
    {
      var elf = new LockingElf(i, eBuffer, santaHandle, maxElfs,
          elfWaiting, elfDone);
      elf.Start();
    }
  }
}

public class LockingSanta : IStartable
{
  private readonly Queue<LockingReindeer> _rBuffer;
  private readonly Queue<LockingElf> _eBuffer;
  private readonly SemaphoreSlim _santaHandle;
  private readonly SemaphoreSlim _sleigh;
  private readonly SemaphoreSlim _warmingHut;
  private readonly SemaphoreSlim _reindeerDone;
  private readonly SemaphoreSlim _elfsWaiting;
  private readonly SemaphoreSlim _elfsDone;

  public LockingSanta(Queue<LockingReindeer> rBuffer,
      Queue<LockingElf> eBuffer, SemaphoreSlim santaHandle,
           SemaphoreSlim sleigh, SemaphoreSlim warmingHut,
               SemaphoreSlim reindeerDone, SemaphoreSlim
               elfsWaiting, SemaphoreSlim elfsDone)
  {
    _rBuffer = rBuffer;
    _eBuffer = eBuffer;
    _santaHandle = santaHandle;
    _sleigh = sleigh;
    _warmingHut = warmingHut;
    _reindeerDone = reindeerDone;
    _elfsWaiting = elfsWaiting;
    _elfsDone = elfsDone;
  }

  public Task Start()
  {
    return Task.Run(() =>
    {
      while (true)
      {
        //Santa is resting
        _santaHandle.Wait();

        var wakeState = WakeState.ElfsIncompetent;

        lock (_rBuffer)
        {
          if (_rBuffer.Count == SCStats.NR_REINDEER)
          {
            wakeState = WakeState.ReindeerBack;
          }
        }

        switch (wakeState)
        {
          case WakeState.ReindeerBack:
            Console.WriteLine("All reindeers are back!");

            //Release reindeers from warming hut
            _warmingHut.Release(SCStats.NR_REINDEER);

            //Setup the sleigh
            _sleigh.Release(SCStats.NR_REINDEER);

            //Deliver presents
            Console.WriteLine("Santa delivering presents");
            Thread.Sleep(100);

            //Release reindeer
            _rBuffer.Clear();
            _reindeerDone.Release(SCStats.NR_REINDEER);
            Console.WriteLine("Reindeer released");
            break;
          case WakeState.ElfsIncompetent:
            Console.WriteLine("3 elfs at the door!");

            _elfsWaiting.Release(SCStats.MAX_ELFS);

            //Answer questions
            Thread.Sleep(100);

            //Back to work incompetent elfs!
            _eBuffer.Clear();
            _elfsDone.Release(SCStats.MAX_ELFS);

            Console.WriteLine("Elfs helped");
            break;
        }
      }
    });
  }
}

public class LockingElf : IStartable
{
  private readonly Random _randomGen = new
      Random(Guid.NewGuid().GetHashCode());
  public int ID { get; private set; }
  private readonly Queue<LockingElf> _buffer;
  private readonly SemaphoreSlim _maxElfs;
  private readonly SemaphoreSlim _santaHandle;
  private readonly SemaphoreSlim _waitingToAsk;
  private readonly SemaphoreSlim _doneAsking;

  public LockingElf(int id, Queue<LockingElf> buffer,
      SemaphoreSlim santaHandle, SemaphoreSlim maxElfs,
      SemaphoreSlim waitingToAsk, SemaphoreSlim doneWaiting)
  {
    _buffer = buffer;
    ID = id;
    _maxElfs = maxElfs;
    _santaHandle = santaHandle;
    _waitingToAsk = waitingToAsk;
    _doneAsking = doneWaiting;
  }

  public Task Start()
  {
    return Task.Run(() =>
    {
      while (true)
      {
        Thread.Sleep(100 * _randomGen.Next(21));

        //Only a fixed amount of elfs can go to santa at a time
        _maxElfs.Wait();

        lock (_buffer)
        {
          _buffer.Enqueue(this);
          if (_buffer.Count == SCStats.MAX_ELFS)
          {
            _santaHandle.Release();
          }
        }

        Console.WriteLine("Elf {0} at the door", ID);

        //Wait for santa to be ready
        _waitingToAsk.Wait();

        //Asking questions
        _doneAsking.Wait();

        //Allow a new elf to visit santa
        _maxElfs.Release();
      }
    });
  }

  public void AskQuestion()
  {
    _waitingToAsk.Release();
  }
}

public class LockingReindeer : IStartable
{
  private readonly Random _randomGen = new
      Random(Guid.NewGuid().GetHashCode());
  public int ID { get; private set; }

  private readonly Queue<LockingReindeer> _reindeerBuffer;
  private readonly SemaphoreSlim _santaHandle;
  private readonly SemaphoreSlim _sleigh;
  private readonly SemaphoreSlim _doneDelivering;
  private readonly SemaphoreSlim _warmingHut;

  public LockingReindeer(int id, Queue<LockingReindeer> buffer,
      SemaphoreSlim santaHandle, SemaphoreSlim sleigh,
      SemaphoreSlim warmingHut, SemaphoreSlim doneDelivering)
  {
    ID = id;
    _reindeerBuffer = buffer;
    _santaHandle = santaHandle;
    _sleigh = sleigh;
    _warmingHut = warmingHut;
    _doneDelivering = doneDelivering;
  }

  public Task Start()
  {
    return Task.Run(() =>
    {
      while (true)
      {
        //Tan on the beaches in the Pacific until Chistmas is
            close
        Thread.Sleep(100 * _randomGen.Next(10));

        lock (_reindeerBuffer)
        {
          _reindeerBuffer.Enqueue(this);
          if (_reindeerBuffer.Count == SCStats.NR_REINDEER)
          {
            _santaHandle.Release();
          }
        }

        //Console.WriteLine("Reindeer {0} is back",ID);

        //Block early arrivals
        _warmingHut.Wait();

        //Wait for santa to be ready
        _sleigh.Wait();

        //Delivering presents

        //Wait for delivery to be done
        _doneDelivering.Wait();
        //Head back to Pacific islands
      }
    });
  }
}
```

C.1.3 Concurrent Queue

The lock based queue implementation is based on Michael L Scott's lock-based queue algorithm described in [19]. The implementation uses two locks, as seen on line 17 and 26, to protect the tail and head of the queue respectively. The first node in the queue is a dummy node which allows the enqueue and dequeue operations to only operate on the tail and head respectively, thereby allowing enqueuing and dequeuing to occur concurrently. Only a single enqueue or dequeue operation can however execute at a time, due to the use of locking. The dummy node is created by the queue's constructor on line 10.

Listing C.3: Lock Based Concurrent Queue Implementation

```
 1 public class Queue<T> : IQueue<T>
 2 {
 3   protected readonly object HeadLock = new object();
 4   protected readonly object TailLock = new object();
 5   private Node _head;
 6   private Node _tail;
 7
 8   public Queue()
 9   {
10     _head = new Node(default(T));
```

```
    _tail = _head;
  }

  public void Enqueue(T item)
  {
    var node = new Node(item);
    lock (TailLock)
    {
      _tail.Next = node;
      _tail = node;
    }
  }

  public bool Dequeue(out T item)
  {
    lock (HeadLock)
    {
      var newHead = _head.Next;
      if (newHead == null)
      {
        item = default(T);
        return false;
      }

      _head = newHead;
      item = newHead.Value;
      return true;
    }
  }

  private class Node
  {
    public Node Next { get; set; }
    public T Value { get; private set; }

    public Node(T value)
    {
      Value = value;
    }
  }
}
```

C.1.4 Concurrent Hashmap

The lock-based concurrent hashmap implementation is based on the concept of lock striping. Lock striping allows L locks to protect B buckets where $B \geq L$ and $B \bmod L = 0$ so that each lock l protects each bucket b where $indexof(b) = indexof(l) \bmod L$[53, p. 304]. When the number of buckets is doubled as a result of resizing the hashmap the same lock protects a particular bucket before and after the resize completes. The buckets protected by a particular lock are referred to as a stripe and multiple stripes can be accessed concurrently.

Along with the backing array of buckets defined on line 49 an array of objects representing the locks is defined on line 48. Before a thread access a bucket it must acquire the lock on the stripe containing the bucket in question. As an example, the `Add` method defined on line 195 calculates the index of the lock to acquire on line 198 before adding the item to the bucket. The `ResizeIfNeeded` method defined on line 287 acquires the lock on all stripes on line 291 ensuring that no other threads are accessing the backing array while being resized. The condition for resizing is checked before all locks are acquired on line 289 and again after all locks have been acquired on line 294, ensuring that only a single thread resizes the backing array for every instance where a resize is required. The object defined on line 47 protect the `_size` variable defined on line 50. This is required as multiple threads can add or remove items to different stripes simultaneously requiring their access to the `_size` variable to be synchronized.

Listing C.4: Lock Based Concurrent Hashmap Implementation

```
1  public abstract class BaseHashMap<K,V> : IMap<K,V>
2  {
3    protected const int DefaultNrBuckets = 16;
4    protected const double LoadFactor = 0.75D;
5
6    public abstract bool ContainsKey(K key);
7    public abstract V Get(K key);
8    public abstract void Add(K key, V value);
9    public abstract bool AddIfAbsent(K key, V value);
10   public abstract bool Remove(K k);
11   public abstract V this[K key] { get; set; }
12   public virtual int Count {  get; protected set; }
13
14   protected int CalculateThreshold(int nrBuckets)
15   {
16     return (int)(nrBuckets * LoadFactor);
17   }
18
19   protected int GetHashCode(K key)
20   {
21     var hashCode = key.GetHashCode();
22     return hashCode < 0 ? 0 - hashCode : hashCode;
23   }
24
25   protected int GetBucketIndex(int length, K key)
26   {
27     return GetBucketIndex(length, GetHashCode(key));
28   }
29
30   protected int GetBucketIndex(int length, int hashCode)
31   {
32     return hashCode % length;
33   }
34
```

```
  public abstract IEnumerator<KeyValuePair<K, V>> GetEnumerator();

  IEnumerator IEnumerable.GetEnumerator()
  {
    return GetEnumerator();
  }
}

public class LockingHashMap<K,V> : BaseHashMap<K,V>
{
  private const int DefaultNrLocks = DefaultNrBuckets;

  private readonly object _sizeLock = new object();
  private readonly object[] _locks;
  private Node[] _buckets;
  private int _size;
  private int _threshold;

  public LockingHashMap() : this(DefaultNrBuckets){}

  public LockingHashMap(int size) : this(size, DefaultNrLocks){}

  private LockingHashMap(int size, int nrLocks)
  {
    if (size % nrLocks != 0)
    {
      throw new Exception("The intital size % nrlocks must be
          equal to zero");
    }
    _buckets = MakeBuckets(size);
    _locks = MakeLocks(nrLocks);
    _threshold = CalculateThreshold(size);
  }


  private Node[] MakeBuckets(int nrBuckets)
  {
    return new Node[nrBuckets]; ;
  }

  private object[] MakeLocks(int nrLocks)
  {
    var temp = new object[nrLocks];
    for (var i = 0; i < nrLocks; i++)
    {
      temp[i] = new object();
    }

    return temp;
  }

  #region Utility

  private int GetLockIndex(int hashCode)
```

```
{
  return hashCode % _locks.Length;
}

private Node CreateNode(K key, V value)
{
  return new Node(key, value);
}

private int GetBucketIndex(int hashCode)
{
  return GetBucketIndex(_buckets.Length, hashCode);
}

private Node FindNode(Node node, K key)
{
  while (node != null && !key.Equals(node.Key))
    node = node.Next;
  return node;
}

private void LockAll()
{
  foreach (var lo in _locks)
  {
    Monitor.Enter(lo);
  }
}

private void UnlockAll()
{
  foreach (var lo in _locks)
  {
    Monitor.Exit(lo);
  }
}

private void InsertInBucket(Node[] buckets, Node node, int
    index)
{
  InsertInBucket(buckets, node, buckets[index], index);
}

private void InsertInBucket(Node node, int index)
{
  InsertInBucket(node, _buckets[index], index);
}

private void InsertInBucket(Node node, Node curNode, int index)
{
  InsertInBucket(_buckets, node, curNode, index);
}

private void InsertInBucket(Node[] buckets, Node node, Node
    curNode, int index)
```

```
{
  if (curNode != null)
  {
    node.Next = curNode;
  }
  buckets[index] = node;
}

#endregion Utility

public override int Count {
  get {
    lock (_sizeLock)
    {
      return _size;
    }
  }

protected set {
    lock (_sizeLock)
    {
      _size = value;
    }
  }
}

public override bool ContainsKey(K key)
{
  var hashCode = GetHashCode(key);
  lock (_locks[GetLockIndex(hashCode)])
  {
    var bucket = _buckets[GetBucketIndex(hashCode)];
    return FindNode(bucket, key) != null;
  }
}

public override V Get(K key)
{
  var hashCode = GetHashCode(key);
  lock (_locks[GetLockIndex(hashCode)])
  {
    var bucket = _buckets[GetBucketIndex(hashCode)];
    var node = FindNode(bucket, key);

    if (node == null)
    {
      //If node is null, key is not in map
      throw new KeyNotFoundException("Key not found. Key:
          "+key);
    }

    return node.Value;
  }
}
```

```

public override void Add(K key, V value)
{
  var hashCode = GetHashCode(key);
  lock (_locks[GetLockIndex(hashCode)])
  {
    var index = GetBucketIndex(hashCode);
    var bucket = _buckets[index];
    var node = FindNode(bucket, key);

    if (node != null)
    {
      //If node is not null, key exist in map. Update the value
      node.Value = value;
    }
    else
    {
      //Else insert the node
      InsertInBucket(CreateNode(key, value),bucket,index);
      lock (_sizeLock)
      {
        _size++;
      }
    }
  }

  ResizeIfNeeded();
}

public override bool AddIfAbsent(K key, V value)
{
  var hashCode = GetHashCode(key);
  lock (_locks[GetLockIndex(hashCode)])
  {
    var index = GetBucketIndex(hashCode);
    var bucket = _buckets[index];
    var node = FindNode(bucket, key);

    if (node != null) return false;
    //If node is not in map insert new node
    InsertInBucket(CreateNode(key, value), bucket, index);
    lock (_sizeLock)
    {
      _size++;
    }
    ResizeIfNeeded();
    return true;
  }
}

public override bool Remove(K key)
{
    var hashCode = GetHashCode(key);
    lock (_locks[GetLockIndex(hashCode)])
```

```
      {
        var index = GetBucketIndex(hashCode);
        var bucket = _buckets[index];
        return RemoveNode(key, bucket, index);
      }
  }

  private bool RemoveNode(K key, Node node, int index)
  {
      if (node == null)
      {
        return false;
      }

      if (node.Key.Equals(key))
      {
        lock (_sizeLock)
        {
          _size--;
        }
        buckets[index] = node.Next;
        return true;
      }


      while (node.Next != null && !key.Equals(node.Next.Key))
        node = node.Next;

      //node.Next == null || node.Next.Key == key
      if (node.Next == null) return false;

      lock (_sizeLock)
      {
        _size--;
      }
      node.Next = node.Next.Next;
      return true;
  }

  private void ResizeIfNeeded()
  {
    if (ResizeCondtion())
    {
        LockAll();
        try
        {
          if (!ResizeCondtion())
          {
            return;
          }
          //Construct new backing array
          var newBucketSize = _buckets.Length * 2;
          var newBuckets = MakeBuckets(newBucketSize);

```

```
            //For each key in the map rehash
            for (var i = 0; i < _buckets.Length; i++)
            {
              var node = _buckets[i];
              while (node != null)
              {
                var bucketIndex = GetBucketIndex(newBucketSize,
                    node.Key);
                InsertInBucket(newBuckets,CreateNode(node.Key,node.Value),bucketIndex);
                node = node.Next;
              }
            }

            //Calculate new resize threshold and assign the
                rehashed backing array
            threshold = CalculateThreshold(newBucketSize);
            _buckets = newBuckets;
          }
          finally
          {
            UnlockAll();
          }
      }
    }

    private bool ResizeCondtion()
    {
      lock (_sizeLock)
      {
        return _size >= _threshold;
      }
    }

    public override V this[K key]
    {
      get { return Get(key); }
      set { Add(key,value); }
    }

    public override IEnumerator<KeyValuePair<K, V>> GetEnumerator()
    {
      LockAll();
      try
      {
        var list = new List<KeyValuePair<K,V>>(_size);
        for (var i = 0; i < _buckets.Length; i++)
        {
          var node = _buckets[i];
          while (node != null)
          {
            list.Add(new KeyValuePair<K, V>(node.Key, node.Value));
            node = node.Next;
          }
        }

        return list.GetEnumerator();
      }
      finally
      {
        UnlockAll();
      }
    }

    private class Node
    {
      public K Key { get; private set; }
      public V Value { get; internal set; }
      public Node Next { get; internal set; }

      public Node(K key, V value)
      {
        Key = key;
        Value = value;
      }
    }
  }
```

C.2 STM Library

This section presents the library-based STM implementations used for the evaluation.

C.2.1 Dining Philosophers Problem

The library based STM uses five STM objects of type `TMVar<bool>`, defined on lines 8 to 12, to represent each of the forks on the table. The `StartPhilosopher` method on line 27 is called for each philosopher in order to start a thread with access to the correct forks. The transaction defined on line 33 attempts to acquired both forks in order to begin eating. Implicit conversion is employed on line 35 to read the state of the forks without having to access the `Value` property. The call to `STMSystem.Retry` on line 37 blocks the calling thread until the state of the forks change, if atleast one of the forks is unavailable. The transaction on line 48 simply makes both forks available, after the philosopher has finished eating on lines 44 to 46.

Listing C.5: STM Library Based Dining Philosophers Implementation

```
1 public class DiningPhilosophers
2 {
3   private const int MAX_EAT_COUNT = 1000;
4
5   public static void Start()
6   {
    var eatCounter = new TMInt(0);
    var fork1 = new TMVar<bool>(true);
    var fork2 = new TMVar<bool>(true);
    var fork3 = new TMVar<bool>(true);
    var fork4 = new TMVar<bool>(true);
    var fork5 = new TMVar<bool>(true);

    var t1 = StartPhilosopher(eatCounter, fork1, fork2);
    var t2 = StartPhilosopher(eatCounter, fork2, fork3);
    var t3 = StartPhilosopher(eatCounter, fork3, fork4);
    var t4 = StartPhilosopher(eatCounter, fork4, fork5);
    var t5 = StartPhilosopher(eatCounter, fork5, fork1);

    t1.Join();
    t2.Join();
    t3.Join();
    t4.Join();
    t5.Join();
  }

  private static Thread StartPhilosopher(TMInt eatCounter,
      TMVar<bool> left, TMVar<bool> right)
  {
    var t1 = new Thread(() =>
    {
      while (eatCounter < MAX_EAT_COUNT)
      {
        STMSystem.Atomic(() =>
        {
          if (!left || !right)
          {
            STMSystem.Retry();
          }

          left.Value = false;
          right.Value = false;
        });

        Console.WriteLine("Thread: " +
            Thread.CurrentThread.ManagedThreadId + " eating.");
        Thread.Sleep(100);
        Console.WriteLine("Eat count: " + ++eatCounter);

        STMSystem.Atomic(() =>
        {
          left.Value = true;
          right.Value = true;
        });

        Thread.Sleep(100);
      }
    });

    t1.Start();

    return t1;
  }
}
```

C.2.2 Santa Claus Problem

As shown on line 124, the library-based Santa Claus problem implementation uses a transaction with an associated orelse clause to wake Santa given either the reindeer or elf condition is true true. The calls to `STMSystem.Retry` on lines 128 and 137 causes Santa to block if the transaction is executed and non of the conditions are true. If one of the read variable change Santa automatically tests the conditions again. As a result the elfs and reindeer does not have to take an explicit action in order to wake Santa. In fact they need not to even know that Santa exists. A call to `STMSystem.Retry` is used on line 171 to ensure that only three elfs can go to Santa at the same time. Generally, transactions containing calls to `STMSystem.Retry` are used ensure that the elfs and reindeer do not progress to the next state before the required conditions e.g. having delivered the presents, are true. The queue utilized is a transactional queue implementation available through the `STM` library.

Listing C.6: STM Library Based Santa Claus Implementation

```
1  public class SantaClausProblem
2  {
3    public static void Start()
4    {
5      var rBuffer = new Queue<Reindeer>();
6      var eBuffer = new Queue<Elf>();
7      var santa = new Santa(rBuffer,eBuffer);
8      santa.Start();
9
10     for (int i = 0; i < SCStats.NR_REINDEER ; i++)
11     {
12       var reindeer = new Reindeer(i, rBuffer);
13       reindeer.Start();
14     }
15
16     for (int i = 0; i < SCStats.NR_ELFS; i++)
17     {
18       var elf = new Elf(i, eBuffer);
19       elf.Start();
20     }
21   }
22 }
23
24 public class Santa : IStartable
25 {
26   private readonly Queue<Reindeer> _rBuffer;
```

```
private readonly Queue<Elf> _eBuffer;

public Santa(Queue<Reindeer> rBuffer, Queue<Elf> eBuffer)
{
  _rBuffer = rBuffer;
  _eBuffer = eBuffer;
}

public Task Start()
{
  return Task.Run(() =>
  {
    while (true)
    {
      var wakestate = SleepUntilAwoken();

      switch (wakestate)
      {
        case WakeState.ReindeerBack:
          HandleReindeer();
          break;
        case WakeState.ElfsIncompetent:
          HandleElfs();
          break;
      }
    }
  });
}

private void HandleElfs()
{
  Console.WriteLine("3 elfs at the door!");
  STMSystem.Atomic(() =>
  {
    foreach (var elf in _eBuffer)
    {
      elf.AskQuestion();
    }
  });

  //Answer questions
  Thread.Sleep(100);

  //Back to work incompetent elfs!
  STMSystem.Atomic(() =>
  {
    for (int i = 0; i < SCStats.MAX_ELFS; i++)
    {
      var elf = _eBuffer.Dequeue();
      elf.BackToWork();
    }
  });

  Console.WriteLine("Elfs helped");
```

```
}

private void HandleReindeer()
{
  Console.WriteLine("All reindeer are back!");

  //Call reindeer from the warming hut
  STMSystem.Atomic(() =>
  {
    foreach (var reindeer in _rBuffer)
    {
      reindeer.CallToSleigh();
    }
  });

  //Setup the sleigh
  STMSystem.Atomic(() =>
  {
    foreach (var reindeer in _rBuffer)
    {
      reindeer.HelpDeliverPresents();
    }
  });

  //Deliver presents
  Console.WriteLine("Santa delivering presents");
  Thread.Sleep(100);

  //Release reindeer
  STMSystem.Atomic(() =>
  {
    while (_rBuffer.Count != 0)
    {
      var reindeer = _rBuffer.Dequeue();
      reindeer.ReleaseReindeer();
    }
  });

  Console.WriteLine("Reindeer released");
}

private WakeState SleepUntilAwoken()
{
  return STMSystem.Atomic(() =>
  {
    if (_rBuffer.Count != SCStats.NR_REINDEER)
    {
      STMSystem.Retry();
    }

    return WakeState.ReindeerBack;
  },
    () =>
    {
          if (_eBuffer.Count != SCStats.MAX_ELFS)
          {
            STMSystem.Retry();
          }

          return WakeState.ElfsIncompetent;
        });
    }
}

public class Elf : IStartable
{
  private readonly Random _randomGen = new
      Random(Guid.NewGuid().GetHashCode());
  public int ID { get; private set; }
  private readonly Queue<Elf> _buffer;
  private readonly TMVar<bool> _waitingToAsk = new
      TMVar<bool>(false);
  private readonly TMVar<bool> _questionAsked = new
      TMVar<bool>(false);

  public Elf(int id, Queue<Elf> buffer)
  {
    _buffer = buffer;
    ID = id;
  }

  public Task Start()
  {
    return Task.Run(() =>
    {
      while (true)
      {
        Thread.Sleep(100 * _randomGen.Next(21));

        STMSystem.Atomic(() =>
        {
          if (_buffer.Count == SCStats.MAX_ELFS)
          {
            STMSystem.Retry();
          }

          _buffer.Enqueue(this);
          _waitingToAsk.Value = true;
        });

        Console.WriteLine("Elf {0} at the door",ID);
        //Waiting on santa
        STMSystem.Atomic(() =>
        {
          if (_waitingToAsk)
          {
            STMSystem.Retry();
          }
        });

        //Asking question

        //Done asking
        STMSystem.Atomic(() =>
        {
          if (!_questionAsked)
          {
            STMSystem.Retry();
          }

          _questionAsked.Value = false;
        });

      }
    });
  }

  public void AskQuestion()
  {
    _waitingToAsk.Value = false;
  }

  public void BackToWork()
  {
    _questionAsked.Value = true;
  }
}

  public class Reindeer : IStartable
{
  private readonly Random _randomGen = new
      Random(Guid.NewGuid().GetHashCode());
  public int ID { get; private set; }

  private readonly Queue<Reindeer> _reindeerBuffer;
  private readonly TMVar<bool> _workingForSanta = new
      TMVar<bool>(false);
  private readonly TMVar<bool> _waitingAtSleigh = new
      TMVar<bool>(false);
  private readonly TMVar<bool> _waitingInHut = new
      TMVar<bool>(false);

  public Reindeer(int id, Queue<Reindeer> buffer)
  {
    ID = id;
    _reindeerBuffer = buffer;
  }

  public Task Start()
  {
    return Task.Run(() =>
    {
```

```
      while (true)
      {
        Thread.Sleep(100 * _randomGen.Next(10));

        STMSystem.Atomic(() =>
        {
          _reindeerBuffer.Enqueue(this);
          _waitingInHut.Value = true;
        });

        Console.WriteLine("Reindeer {0} is back",ID);

        //Waiting in the warming hut
        STMSystem.Atomic(() =>
        {
          if (_waitingInHut)
          {
            STMSystem.Retry();
          }
        });

        //Wait for santa to be ready
        STMSystem.Atomic(() =>
        {
          if (_waitingAtSleigh)
          {
            STMSystem.Retry();
          }
        });

        //Delivering presents

        //Wait to be released by santa
        STMSystem.Atomic(() =>
        {
          if (_workingForSanta)
          {
            STMSystem.Retry();
          }
        });
      }
    });
  }

  public void CallToSleigh()
  {
    STMSystem.Atomic(() =>
    {
      _waitingInHut.Value = false;
      _waitingAtSleigh.Value = true;
    });
  }

  public void HelpDeliverPresents()
  {
    STMSystem.Atomic(() =>
    {
      _waitingAtSleigh.Value = false;
      _workingForSanta.Value = true;
    });
  }

  public void ReleaseReindeer()
  {
    _workingForSanta.Value = false;
  }
}
```

C.2.3 Concurrent Queue

The STM concurrent queue implementation is implemented as a linked list of nodes. As with the lock-based concurrent queue implementation, the library-based STM concurrent queue implementation uses a dummy node, allowing the enqueue and dequeue operations to operate only on the tail and head respectively. The _head and _tail, defined on lines 3 and 4, are of type TMVar<Node> allowing the STM system to track assignments to the encapsulated variables. Similarly the Next field of the Node class defined on line 40 is of type TMVar<Node>. The Dequeue method defined in line 24, skips over the dummy item when reading the node to remove from the queue on line 28. The call to STMSystem.Retry on line 32 causes a thread calling Dequeue on an empty queue to block until the queue is no longer empty. The assignment on line 32 sets the node representing the item to dequeue as the new dummy node, as it is no longer needed and already points to the next item in the queue.

Listing C.7: STM Library Based Concurrent Queue Implementation

```
1  public class Queue<T>
2  {
3    private readonly TMVar<Node> _head = new TMVar<Node>(null);
4    private readonly TMVar<Node> _tail = new TMVar<Node>(null);
5
6    public Queue()
7    {
8      var node = new Node(default(T));
9      _head.Value = node;
10     _tail.Value = node;
11   }
12
13   public void Enqueue(T value)
14   {
15     STMSystem.Atomic(() =>
16     {
17       var node = new Node(value);
```

```
      var curTail = _tail.Value;
      curTail.Next.Value = node;
      _tail.Value = node;
    });
  }

  public T Dequeue()
  {
    return STMSystem.Atomic(() =>
    {
      var node = _head.Value.Next.Value;

      if (node == null)
      {
        STMSystem.Retry();
      }

      _head.Value = node;
      return node.Value;
    });
  }

  private class Node
  {
    public readonly TMVar<Node> Next = new TMVar<Node>(null);
    public readonly T Value;

    public Node(T value)
    {
      Value = value;
    }
  }
}
```

C.2.4 Concurrent Hashmap

The library-based STM hashmap defines the collision list as a linked list of instances of the `Node` class defined on line 243. The backing array, defined on line 4, of type `TMVar<TMVar<Node>[]>`, that is a transactional variable to an array of transactional variables to instances of the node `Node` class. This allows the STM system to track the assignment of a new backing array as a result of resizing the hashmap as well as track the assignment of the first node in each bucket trough the `TMVars` in the backing array. As the `Next` property of the `Node` class, defined on line 247, is type `TMVar<Node>` the STM system is able to track changes to the collision list. As the `Add` method defined on line 97 updates the value of a node if the key to add is already present in the hashmap, the `Value` property of the `Node` class is of type `TMVar<V>` where `V` is the type parameter, defining the type of the values added to the hashmap. The `_size` variable is of type `TMInt` and is increment and decremented using the supplied ++ and -- overloads defined in the STM library.

Listing C.8: STM Library Based Concurrent Hashmap Implementation

```
public class STMHashMap<K,V> : BaseHashMap<K,V>
{
   //TMVar to (array of TMVars to Node )
  private readonly TMVar<TMVar<Node>[]> _buckets = new
      TMVar<TMVar<Node>[]>();
  private readonly TMInt _threshold = new TMInt();
  private TMInt _size = new TMInt();

  public STMHashMap() : this(DefaultNrBuckets)
  {

  }

  public STMHashMap(int nrBuckets)
  {
    _buckets.Value = MakeBuckets(nrBuckets);
    _threshold.Value = CalculateThreshold(nrBuckets);
  }

  /// <summary>
  /// Creates and initializes the backing array
  /// </summary>
  /// <param name="nrBuckets"></param>
  /// <returns></returns>
  private TMVar<Node>[] MakeBuckets(int nrBuckets)
  {
    var temp = new TMVar<Node>[nrBuckets];
    for (var i = 0; i < nrBuckets; i++)
    {
      temp[i] = new TMVar<Node>();
    }

    return temp;
  }


  #region Utility

  private Node CreateNode(K key, V value)
  {
    return new Node(key,value);
  }

  private int GetBucketIndex(K key)
  {
    return GetBucketIndex(_buckets.Value.Length, key);
  }

  private Node FindNode(K key)
  {
    return FindNode(key, GetBucketIndex(key));
  }
```

```

private Node FindNode(K key, int bucketIndex)
{
  return FindNode(key, _buckets.Value[bucketIndex].Value);
}

private Node FindNode(K key, Node node)
{
  while (node != null && !key.Equals(node.Key))
    node = node.Next.Value;
  return node;
}

private void InsertInBucket(TMVar<Node> bucketVar, Node node)
{
  var curNode = bucketVar.Value;
  if (curNode != null)
  {
    node.Next.Value = curNode;
  }
  bucketVar.Value = node;
}

#endregion Utility

public override bool ContainsKey(K key)
{
  return STMSystem.Atomic(() => FindNode(key) != null);
}

public override V Get(K key)
{
  return STMSystem.Atomic(() =>
  {
    var node = FindNode(key);
    if (node == null)
    {
      //If node == null key is not present in dictionary
      throw new KeyNotFoundException("Key not found. Key: " +
          key);
    }

    return node.Value.Value;
  });
}

public override void Add(K key, V value)
{
  STMSystem.Atomic(() =>
  {
    var bucketIndex = GetBucketIndex(key);
    //TMVar wrapping the immutable chain list
    var bucketVar = _buckets.Value[bucketIndex];
    var node = FindNode(key, bucketVar.Value);
```

```

      if (node != null)
      {
        //If node is not null key exist in map. Update the value
        node.Value.Value = value;
      }
      else
      {
        //Else insert the node
        InsertInBucket(bucketVar, CreateNode(key,value));
        _size++;
        ResizeIfNeeded();
      }
    });
  }

  public override bool AddIfAbsent(K key, V value)
  {
    return STMSystem.Atomic(() =>
    {
      var bucketIndex = GetBucketIndex(key);
      //TMVar wrapping the immutable chain list
      var bucketVar = _buckets.Value[bucketIndex];
      var node = FindNode(key, bucketVar.Value);

      if (node == null)
      {
        //If node is not found key does not exist so insert
        InsertInBucket(bucketVar, CreateNode(key,value));
        _size++;
        ResizeIfNeeded();
        return true;
      }

      return false;
    });
  }

  private void ResizeIfNeeded()
  {
    if (_size.Value >= _threshold.Value)
    {
      Resize();
    }
  }

  private void Resize()
  {
    //Construct new backing array
    var newBucketSize = _buckets.Value.Length * 2;
    var newBuckets = MakeBuckets(newBucketSize);

    //For each key in the map rehash
    for (var i = 0; i < _buckets.Value.Length; i++)
    {
      var bucket = _buckets.Value[i];
      var node = bucket.Value;
      while (node != null)
      {
        var bucketIndex = GetBucketIndex(newBucketSize, node.Key);
        InsertInBucket(newBuckets[bucketIndex],
            CreateNode(node.Key, node.Value));
        node = node.Next.Value;
      }
    }

    //Calculate new resize threshold and assign the rehashed
        backing array
    _threshold.Value = CalculateThreshold(newBucketSize);
    _buckets.Value = newBuckets;
  }

  public override bool Remove(K key)
  {
    return STMSystem.Atomic(() =>
    {
      var bucketIndex = GetBucketIndex(key);
      //TMVar wrapping the immutable chain list
      var bucketVar = _buckets.Value[bucketIndex];
      var firstNode = bucketVar.Value;

      return RemoveNode(key, firstNode, bucketVar);
    });
  }

  private bool RemoveNode(K key, Node node, TMVar<Node> bucketVar)
  {
    if (node == null)
    {
      return false;
    }

    if (node.Key.Equals(key))
    {
      _size--;
      bucketVar.Value = node.Next;
      return true;
    }

    while (node.Next != null && !key.Equals(node.Next.Value.Key))
      node = node.Next.Value;

    //node.Next == null || node.Next.Key == key
    if (node.Next == null) return false;

    _size--;
    node.Next.Value = node.Next.Value.Next;
    return true;
  }

  public override IEnumerator<KeyValuePair<K, V>> GetEnumerator()
  {
    return STMSystem.Atomic(() =>
    {
      var list = new List<KeyValuePair<K, V>>(_size.Value);
      for (var i = 0; i < _buckets.Value.Length; i++)
      {
        var bucket = _buckets.Value[i];
        var node = bucket.Value;
        while (node != null)
        {
          list.Add(new KeyValuePair<K, V>(node.Key, node.Value));
          node = node.Next.Value;
        }
      }
      return list.GetEnumerator();
    });
  }

  public override V this[K key]
  {
    get { return Get(key); }
    set { Add(key, value); }
  }

  public override int Count
  {
    get { return _size.Value; }
  }

  private class Node
  {
    public K Key { get; private set; }
    public TMVar<V> Value { get; private set; }
    public TMVar<Node> Next { get; private set; }

    public Node(K key, V value)
    {
      Key = key;
      Value = new TMVar<V>(value);
      Next = new TMVar<Node>();
    }
  }
}
```

C.3 AC#

This section presents the AC# implementations used for the evaluation. As the strategies employed for the AC# implementations are similar to those of the library-based STM implementations the descriptions focus on the differences between the implementations as well as how the abstractions provided by AC# are utilized.

C.3.1 Dining Philosophers Problem

The AC# dining philosophers implementation represents the forks as instances of the `Fork` class defined on line 58. Five forks are created on lines 8 to 12 and passed to the `StartPhilosopher` method on lines 14 to 18. The transaction defined in line 33 acquires both forks by calling the forks `AttemptToPickUp` which is itself defined using a transaction. If a fork is unavailable the calling thread will block until the state changes as a result of the retry statement on line 74. The transaction defined on line 43 puts down both forks after eating on lines 39 to 41 has finished.

Listing C.9: AC# Based Dining Philosophers Implementation

```
1  public class DiningPhilosopher
2  {
3    private static readonly int MAX_EAT_COUNT = 1000;
4    private static atomic int eatCounter = 0;
5
6    public void Start()
7    {
8      var fork1 = new Fork();
9      var fork2 = new Fork();
10     var fork3 = new Fork();
11     var fork4 = new Fork();
12     var fork5 = new Fork();
13
14     var t1 = StartPhilosopher(fork1, fork2);
15     var t2 = StartPhilosopher(fork2, fork3);
16     var t3 = StartPhilosopher(fork3, fork4);
17     var t4 = StartPhilosopher(fork4, fork5);
18     var t5 = StartPhilosopher(fork5, fork1);
19
20     t1.Join();
21     t2.Join();
22     t3.Join();
23     t4.Join();
24     t5.Join();
25   }
26
27   private Thread StartPhilosopher(Fork left, Fork right)
28   {
29     var t1 = new Thread(() =>
30     {
31       while (eatCounter < MAX_EAT_COUNT)
32       {
33         atomic
34         {
35           left.AttemptToPickUp();
36           right.AttemptToPickUp();
```

```
        }

        Console.WriteLine("Thread: " +
            Thread.CurrentThread.ManagedThreadId + " eating.");
        Thread.Sleep(100);
        Console.WriteLine("Eat count: " + ++eatCounter);

        atomic
        {
          left.PutDown();
          right.PutDown();
        }

        Thread.Sleep(100);
      }
    });

    t1.Start();

    return t1;
  }

  public class Fork
  {
    private atomic
    private bool State { get; set; }

    public Fork()
    {
      State = true;
    }

    public void AttemptToPickUp()
    {
      atomic
      {
        if (!State)
        {
          retry;
        }

        State = false;
      }
    }

    public void PutDown()
    {
      atomic
      {
        State = true;
      }
    }
  }
}
```

C.3.2 Santa Claus Problem

As with the library-based Santa Claus implementation AC# uses an `atomic` block with an associated `orelse` block, defined on line 65, to wake Santa when either the reindeer or elf condition are true. On line 178 the `retry` statement is used to ensure that no more than three elfs can go to Santa at any given point. Similarly, transactions utilizing `retry` statement are us ensure that the elfs and reindeer do not proceeded to next state before the required condition is true throughout the implementation.

Listing C.10: AC# Based Santa Claus Implementation

```
1  public class SantaClausProblem
2  {
3    public const int NR_REINDEER = 9;
4    public const int NR_ELFS = 6;
5    public const int MAX_ELFS = 3;
6
7    public static void Main()
8    {
9      var rBuffer = new Queue<Reindeer>();
10     var eBuffer = new Queue<Elf>();
11     var santa = new Santa(rBuffer, eBuffer);
12     santa.Start();
13
14     for (int i = 0; i < SantaClausProblem.NR_REINDEER; i++)
15     {
16       var reindeer = new Reindeer(i, rBuffer);
17       reindeer.Start();
18     }
19
20     for (int i = 0; i < SantaClausProblem.NR_ELFS; i++)
21     {
22       var elf = new Elf(i, eBuffer);
23       elf.Start();
24     }
25
26     System.Console.WriteLine("Press any key to terminate...");
27     System.Console.ReadKey();
28   }
29 }
30
31 public class Santa
32 {
33   private readonly Queue<Reindeer> _rBuffer;
34   private readonly Queue<Elf> _eBuffer;
35
36   public Santa(Queue<Reindeer> rBuffer, Queue<Elf> eBuffer)
37   {
38     _rBuffer = rBuffer;
39     _eBuffer = eBuffer;
40   }
41
```

```
    public Task Start()
  {
    return Task.Run(() =>
    {
      while (true)
      {
        var wakestate = SleepUntilAwoken();

        switch (wakestate)
        {
          case WakeState.ReindeerBack:
            HandleReindeer();
            break;
          case WakeState.ElfsIncompetent:
            HandleElfs();
            break;
        }
      }
    });
  }

  private WakeState SleepUntilAwoken()
  {
    atomic
    {
      if (_rBuffer.Count != SantaClausProblem.NR_REINDEER)
      {
        retry;
      }

      return WakeState.ReindeerBack;
    }
    orelse
    {
      if (_eBuffer.Count != SantaClausProblem.MAX_ELFS)
      {
        retry;
      }

      return WakeState.ElfsIncompetent;
    }
  }

  private void HandleReindeer()
  {
    Console.WriteLine("All reindeer are back!");

          //Call reindeer from the warming hut
    atomic
    {
      foreach (var reindeer in _rBuffer)
      {
        reindeer.CallToSleigh();
      }
    }

    //Setup the sleigh
    atomic
    {
      foreach (var reindeer in _rBuffer)
      {
        reindeer.HelpDeliverPresents();
      }
    }

    //Deliver presents
    Console.WriteLine("Santa delivering presents");
    Thread.Sleep(100);

    //Release reindeer
    atomic
    {
      while (_rBuffer.Count != 0)
      {
        var reindeer = _rBuffer.Dequeue();
        reindeer.ReleaseReindeer();
      }
    }

    Console.WriteLine("Reindeer released");
  }

  private void HandleElfs()
  {
    Console.WriteLine("3 elfs at the door!");
    atomic
    {
      foreach (var elf in _eBuffer)
      {
        elf.AskQuestion();
      }
    }

    //Answer questions
    Thread.Sleep(100);

    //Back to work incompetent elfs!
    atomic
    {
      for (int i = 0; i < SantaClausProblem.MAX_ELFS; i++)
      {
        var elf = _eBuffer.Dequeue();
        elf.BackToWork();
      }
    }

    Console.WriteLine("Elfs helped");
  }
```

```
}

public class Elf
{
  private Random randomGen = new
      Random(Guid.NewGuid().GetHashCode());
  public int ID { get; private set; }
  private Queue<Elf> _buffer;
  private atomic bool _waitingToAsk = false;
  private atomic bool _questionAsked = false;

  public Elf(int id, Queue<Elf> buffer)
  {
    _buffer = buffer;
    ID = id;
  }

  public Task Start()
  {
    return Task.Run(() =>
    {
      while (true)
      {
        Thread.Sleep(100 * randomGen.Next(21));

        atomic
        {
          if (_buffer.Count == SantaClausProblem.MAX_ELFS)
          {
            retry;
          }

          _buffer.Enqueue(this);
          _waitingToAsk = true;
        }

        Console.WriteLine("Elf {0} at the door", ID);

        //Waiting on santa
        atomic
        {
          if (_waitingToAsk)
          {
            retry;
          }
        }

        //Asking question

        //Done asking
        atomic
        {
          if (!_questionAsked)
          {
            retry;
          }

          _questionAsked = false;
        }
      }
    });
  }

  public void AskQuestion()
  {
    _waitingToAsk = false;
  }

  public void BackToWork()
  {
    _questionAsked = true;
  }
}

public class Reindeer
{
  private readonly Random randomGen = new
      Random(Guid.NewGuid().GetHashCode());
  public int ID { get; private set; }

  private Queue<Reindeer> reindeerBuffer;
  private atomic bool _workingForSanta = false;
  private atomic bool _waitingAtSleigh = false;
  private atomic bool _waitingInHut = false;

  public Reindeer(int id, Queue<Reindeer> buffer)
  {
    ID = id;
    reindeerBuffer = buffer;
  }

  public Task Start()
  {
    return Task.Run(() =>
    {
      while (true)
      {
        Thread.Sleep(100 * randomGen.Next(10));

        atomic
        {
          reindeerBuffer.Enqueue(this);
          _waitingInHut = true;
        }

        Console.WriteLine("Reindeer {0} is back",ID);

                  //Waiting in the warming hut
        atomic
        {
          if (_waitingInHut)
          {
            retry;
          }
        }

        //Wait for santa to be ready
        atomic
        {
          if (_waitingAtSleigh)
          {
            retry;
          }
        }

        //Delivering presents

        //Wait to be released by santa
        atomic
        {
          if (_workingForSanta)
          {
            retry;
          }
        }
      }
    });
  }

      public void CallToSleigh()
  {
    atomic
    {
      _waitingInHut = false;
      _waitingAtSleigh = true;
    }
  }

  public void HelpDeliverPresents()
  {
    atomic
    {
      _waitingAtSleigh = false;
      _workingForSanta = true;
    }

  }

  public void ReleaseReindeer()
  {
    _workingForSanta = false;
  }
```

```
}

public enum WakeState
{
  ReindeerBack,
  ElfsIncompetent
}
```

C.3.3 Concurrent Queue

As with the lock and library-based implementations the AC# implementation is implemented as a linked list of nodes and uses a dummy node. The head and tail nodes, defined on lines 3 and 4, are declared `atomic` along with the `Next` property of the `Node` class defined on 40. The `retry` on line 32 is used to block a thread calling `Dequeue` on an empty queue.

Listing C.11: AC# Based Concurrent Queue Implementation

```
1  public class Queue<T>
2  {
3    private atomic Node _head;
4    private atomic Node _tail;
5
6    public Queue()
7    {
8      var node = new Node(default(T));
9      _head = node;
10     _tail = node;
11   }
12
13   public void Enqueue(T value)
14   {
15     atomic
16     {
17       var node = new Node(value);
18       _tail.Next = node;
19       _tail = node;
20     }
21   }
22
23
24   public T Dequeue()
25   {
26     atomic
27     {
28       var node = _head.Next;
29
30       if (node == null)
31       {
32         retry;
33       }
34
```

```
      _head = node;
      return node.Value;
    }
  }

  private class Node
  {
    public atomic Node Next { get; set; }
    public readonly T Value;

    public Node(T value)
    {
      Value = value;
    }
  }
}
```

C.3.4 Concurrent Hashmap

As with the other concurrent hashmap implementations, the AC# concurrent hashmap implementation utilizes a linked list of nodes for the collision lists. The `Node` class is defined on line 239. As seen on line 243 its next property is declared with the `atomic` modifier, allowing the STM system to detect changes to the list. Similarly the `Node` class's `Value` property is declared with the `atomic` modifier, as a call to the `Add` method with a key already present in the hashmap causes the value of the node representing the key/value pair to be updated. The backing array, seen on line 3, is of type `Bucket[]` and is declared with the atomic modifier. The `atomic` modifier allows the STM system to track the assignment of a new backing array as a result of resizing the hashmap. The `Bucket` class is defined on line 234 and has an `atomic Value` property to which the first node in the buckets linked list is assigned, if such an item exists. Together with the `atomic Next` property of the `Node` class, this allows the STM system to detect any changes to the collision list.

Listing C.12: AC# Based Concurrent Hashmap Implementation

```
1  public class StmHashMap<K,V> : BaseHashMap<K,V>
2  {
3    private atomic Bucket[] _buckets;
4    private atomic int _threshold;
5    private atomic int _size;
6
7    public StmHashMap() : this(DefaultNrBuckets)
8    {
9
10   }
11
12   public StmHashMap(int nrBuckets)
13   {
14     _buckets = MakeBuckets(nrBuckets);
```

```
    _threshold = CalculateThreshold(nrBuckets);
  }

  private Bucket[] MakeBuckets(int nrBuckets)
  {
    var temp = new Bucket[nrBuckets];
    for (int i = 0; i < nrBuckets; i++)
    {
      temp[i] = new Bucket();
    }
    return temp;
  }

  #region Utility

  private Node CreateNode(K key, V value)
  {
    return new Node(key, value);
  }

  private int GetBucketIndex(K key)
  {
    return GetBucketIndex(_buckets.Length, key);
  }

  private Node FindNode(K key)
  {
    return FindNode(key, GetBucketIndex(key));
  }

  private Node FindNode(K key, int bucketIndex)
  {
    return FindNode(key, _buckets[bucketIndex].Value);
  }

  private Node FindNode(K key, Node node)
  {
    while (node != null && !key.Equals(node.Key))
      node = node.Next;
    return node;
  }

  private void InsertInBucket(Bucket bucketVar, Node node)
  {
    var curNode = bucketVar.Value;
    if (curNode != null)
    {
      node.Next = curNode;
    }
    bucketVar.Value = node;
  }

  #endregion Utility

```

```
public override bool ContainsKey(K key)
{
  return FindNode(key) != null;
}

public override V Get(K key)
{
  atomic
  {
    var node = FindNode(key);
    if(node == null)
    {
      //If node == null key is not present in dictionary
      throw new KeyNotFoundException("Key not found. Key: " +
          key);
    }
    return node.Value;
  }
}

public override void Add(K key, V value)
{
  atomic
  {
    var bucketIndex = GetBucketIndex(key);
    //TMVar wrapping the immutable chain list
    var bucketVar = _buckets[bucketIndex];
    var node = FindNode(key, bucketVar.Value);

    if (node != null)
    {
      //If node is not null key exist in map. Update the value
      node.Value = value;
    }
    else
    {
      //Else insert the node
      InsertInBucket(bucketVar, CreateNode(key, value));
      _size++;
      ResizeIfNeeded();
    }
  }
}

public override bool AddIfAbsent(K key, V value)
{
  atomic
  {
    var bucketIndex = GetBucketIndex(key);
    //TMVar wrapping the immutable chain list
    var bucketVar = _buckets[bucketIndex];
    var node = FindNode(key, bucketVar.Value);

    if (node == null)
```

```
      {
        //If node is not found key does not exist so insert
        InsertInBucket(bucketVar, CreateNode(key, value));
        _size++;
        ResizeIfNeeded();
        return true;
      }

      return false;
    }
  }
  private void ResizeIfNeeded()
  {
    if (_size >= _threshold)
    {
      Resize();
    }
  }

  private void Resize()
  {
    //Construct new backing array
    var newBucketSize = _buckets.Length * 2;
    var newBuckets = MakeBuckets(newBucketSize);

    //For each key in the map rehash
    for (var i = 0; i < _buckets.Length; i++)
    {
      var bucket = _buckets[i];
      var node = bucket.Value;
      while (node != null)
      {
        var bucketIndex = GetBucketIndex(newBucketSize, node.Key);
        InsertInBucket(newBuckets[bucketIndex],
            CreateNode(node.Key, node.Value));
        node = node.Next;
      }
    }

    //Calculate new resize threshold and assign the rehashed
        backing array
    _threshold = CalculateThreshold(newBucketSize);
    _buckets = newBuckets;
  }

  public override bool Remove(K key)
  {
    atomic
    {
      var bucketIndex = GetBucketIndex(key);
      //TMVar wrapping the immutable chain list
      var bucketVar = _buckets[bucketIndex];
      var firstNode = bucketVar.Value;

      return RemoveNode(key, firstNode, bucketVar);
    }
  }

  private bool RemoveNode(K key, Node node, Bucket bucketVar)
  {
    if (node == null)
    {
      return false;
    }

    if (node.Key.Equals(key))
    {
      _size--;
      bucketVar.Value = node.Next;
      return true;
    }

    while (node.Next != null && !key.Equals(node.Next.Key))
      node = node.Next;

    //node.Next == null || node.Next.Key == key
    if (node.Next == null) return false;

    _size--;
    node.Next = node.Next.Next;
    return true;
  }

  public override IEnumerator<KeyValuePair<K, V>> GetEnumerator()
  {
    atomic
    {
      var list = new List<KeyValuePair<K, V>>(_size);
      for (var i = 0; i < _buckets.Length; i++)
      {
        var bucket = _buckets[i];
        var node = bucket.Value;
        while (node != null)
        {
          var keyValuePair = new KeyValuePair<K, V>(node.Key,
              node.Value);
          list.Add(keyValuePair);
          node = node.Next;
        }
      }
      return list.GetEnumerator();
    }
  }

  public override V this[K key]
  {
    get { return Get(key); }
    set { Add(key, value); }
  }

  public override int Count
  {
    get { return _size; }
  }

  private class Bucket
  {
    public atomic Node Value { get; set; }
  }

  private class Node
  {
    public K Key { get; private set; }
    public atomic V Value { get; set; }
    public atomic Node Next { get; set; }

    public Node(K key, V value)
    {
      Key = key;
      Value = value;
    }
  }
}
```

D Summary

This master thesis investigates whether language integrated STM is a valid alternative to locking in terms of usability, and provides additional benefits compared to library-based STM. To do so, an extension of C# called AC# was implemented. AC# provides integrated support for STM, including conditional synchronization using the retry and orelse constructs as well as nesting of transactions.

In order to develop AC# a set of requirements were defined, detailing how the underlying STM system should behave in relation to for example tracking granularity, atomicity level and nesting. Based on the requirements AC# was designed. The design includes a description of new language constructs as well as a description of modifications to existing language features. A number of STM implementations were investigated. Based on this investigation as well as the requirements and design, an STM library, utilizing the TLII algorithm, was implemented. The STM library was tested using a number of unit tests, ensuring that transactions are executed correctly. In order to perform the actual integration of STM into C#, the open source Roslyn C# compiler was extended. This required a deep knowledge of the Roslyn project and its structure, which initiated an investigation of Roslyn. Due to the limited availability of literature with regards to Roslyn, much of the knowledge obtained in this area was acquired by reading and debugging Roslyn's source code. A number of unit tests were defined for the Roslyn extension, in order to ensure that all STM constructs and the integration with the existing language features work correctly.

For each of the concurrency approaches: AC#, the STM library and locking in C#, implementations of the Dining Philosophers problem, the Santa Claus problem, a concurrent queue, and a concurrent hashmap were created. These implementations were analyzed according to a set of usability characteristics facilitating a conclusion upon the usability of language integrated STM. Our evaluation concludes that AC# is a valid alternative to locking, and provides better usability than library-based STM.

List of Acronyms

CPU	Central Processing Unit
STM	Software Transactional Memory
DSTM	Dynamic Software Transactional Memory
TL	Threads & Locks
CLR	Common Language Runtime
CIL	Common Intermediate Language
FIFO	First In First Out
OOP	Object Oriented Programming
IO	Input/Output
CAS	Compare-And-Swap
API	Application Programming Interface
JIT	Just-In-Time Compilation
GC	Garbage Collection
VB	Visual Basic
DSL	Domain Specific Language
REPL	Read-Eval-Print Loop
MVCC	Multiversion Concurrency Control
TRP	Transactional Reference Parameter
TOP	Transactional Output Parameter
XML	Extensible Markup Language
LALR	Look Ahead LR

Bibliography

[1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's journal*, vol. 30, no. 3, pp. 202–210, 2005.

[2] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[3] T. Ugleholdt Hansen, A. Pørtner Karlsen, and K. Breinholt Laurberg, *Investigation of trending concurrency models by comparison of performance and characteristics: Software Transactional Memory, Actor Model, and Threads & Locks.* Aalborg University. Department of Computer Science, 2015.

[4] Microsoft. (2015, 2) Roslyn wiki. [Online]. Available: https://github.com/dotnet/roslyn

[5] ——. (2015, 2) Core clr repository. [Online]. Available: https://github.com/dotnet/coreclr

[6] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming.* ACM, 2005, pp. 48–60.

[7] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing.* ACM, 2003, pp. 92–101.

[8] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference.* IEEE, 2003, pp. 522–529.

[9] T. Harris, J. Larus, and R. Rajwar, "Transactional memory," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–263, 2010.

[10] M. Herlihy, V. Luchangco, and M. Moir, "A flexible framework for implementing software transactional memory," in *ACM SIGPLAN Notices*, vol. 41, no. 10. ACM, 2006, pp. 253–262.

[11] T. Harris and K. Fraser, "Language support for lightweight transactions," in *ACM SIGPLAN Notices*, vol. 38, no. 11. ACM, 2003, pp. 388–402.

[12] D. Dice, O. Shalev, and N. Shavit, “Transactional locking ii,” in *Distributed Computing.* Springer, 2006, pp. 194–208.

[13] A. S. Tanenbaum, *Modern operating systems.* Prentice Hall Press, 2008.

[14] J. Duffy, “A (brief) retrospective on transactional memory,” 2010, Located February 2015. [Online]. Available: http://joeduffyblog.com/2010/01/03/a-brief-retrospective-on-transactional-memory/

[15] N. Shavit and D. Touitou, “Software transactional memory,” *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[16] T. Harris, “Exceptions and side-effects in atomic blocks,” *Science of Computer Programming*, vol. 58, no. 3, pp. 325–343, 2005.

[17] M. Corporation, “C# language specification version 5.0,” 2013, Located March 2015. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=7029

[18] Microsoft. (2015, 2) Core clr blog. [Online]. Available: http://blogs.msdn.com/b/dotnet/archive/2015/02/03/coreclr-is-now-open-source.aspx

[19] M. M. Michael and M. L. Scott, “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,” in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing.* ACM, 1996, pp. 267–275.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms.* MIT press, 2009.

[21] R. W. Sebesta, *Concepts of Programming Languages, 10th Edition.* Pearson, 2012, ISBN: 978-0-13-139531-2.

[22] Microsoft, “Overview of synchronization primitives,” Located May 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms228964%28v=vs.110%29.aspx

[23] ——. (2015, May) Monitor class. [Online]. Available: https://msdn.microsoft.com/en-us/library/System.Threading.Monitor.aspx

[24] ——. (2015, May) Monitor class. [Online]. Available: https://msdn.microsoft.com/en-us/library/System.Threading.Mutex.aspx

[25] ——. (2015, May) Monitor class. [Online]. Available: https://msdn.microsoft.com/en-us/library/System.Threading.SemaphoreSlim.aspx

[26] ——. (2015, May) Monitor class. [Online]. Available: https://msdn.microsoft.com/en-us/library/System.Threading.SpinLock.aspx

[27] ——. (2015, May) Monitor class. [Online]. Available: https://msdn.microsoft.com/en-us/library/System.Threading.ReaderWriterLock.aspx

[28] M. Herlihy, "Transactional memory," in *Encyclopedia of Parallel Computing.* Springer, 2011, pp. 2079–2086.

[29] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel, "Committing conflicting transactions in an stm," in *ACM Sigplan Notices*, vol. 44, no. 4. ACM, 2009, pp. 163–172.

[30] K. Ng, M. Warren, P. Golde, and A. Hejlsberg, "The roslyn project, exposing the c# and vb compiler's code analysis," *White paper, Microsoft (Oct. 2011)*, 2012.

[31] M. T. Dustin Campbell, "The future of c#," 2014, build 2014. [Online]. Available: http://channel9.msdn.com/Events/Build/2014/2-577

[32] C. Team, "Roslyn ctp introduces interactive code for c#," 2012, Located March 2015. [Online]. Available: http://blogs.msdn.com/b/csharpfaq/archive/2012/01/30/roslyn-ctp-introduces-interactive-code-for-c.aspx

[33] D. Campbell, "Going deeper with project roslyn: Exposing the c# and vb compiler's code analysis," 2012, lang.NEXT 2012. [Online]. Available: https://channel9.msdn.com/Events/Lang-NEXT/Lang-NEXT-2012/Roslyn

[34] C. N. Fischer, R. K. Cytron, and R. J. LeBlanc, *Crafting a compiler.* Addison-Wesley Publishing Company, 2009.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Pearson Education, 1994.

[36] V. Sadov(VSadov), "Roslyn's performance," 2014, Located March 2015. [Online]. Available: https://roslyn.codeplex.com/discussions/541953

[37] A. D. G. (ADGreen). (2014, 4) Provide access to the binder. [Online]. Available: https://roslyn.codeplex.com/discussions/541303

[38] N. Gafter, "Flow analysis in the roslyn c# compiler," 2011, Located March 2015. [Online]. Available: https://github.com/dotnet/roslyn/blob/master/src/Compilers/CSharp/Portable/FlowAnalysis/Flow%20Analysis%20Design.docx

[39] E. Lippert, "Persistence, facades and roslyn's red-green trees," 2012, Located March 2015. [Online]. Available: http://blogs.msdn.com/b/ericlippert/archive/2012/06/08/persistence-facades-and-roslyn-s-red-green-trees.aspx

[40] Microsoft. (2015, 1) .net compiler platform (”roslyn”) overview. [Online]. Available: https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview

[41] R. Hickey. (2015, March) Concurrent programming. [Online]. Available: http://clojure.org/concurrent_programming

[42] C. J. Rossbach, O. S. Hofmann, and E. Witchel, “Is transactional programming actually easier?” *ACM Sigplan Notices*, vol. 45, no. 5, pp. 47–56, 2010.

[43] V. Pankratius, A.-R. Adl-Tabatabai, and F. Otto, *Does transactional memory keep its promises?: results from an empirical study.* Univ., Fak. für Informatik, 2009.

[44] C. Blundell, E. C. Lewis, and M. M. Martin, “Subtleties of transactional memory atomicity semantics,” *Computer Architecture Letters*, vol. 5, no. 2, 2006.

[45] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott, “Privatization techniques for software transactional memory,” in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing.* ACM, 2007, pp. 338–339.

[46] B. Hindman and D. Grossman, “Atomicity via source-to-source translation,” in *Proceedings of the 2006 workshop on Memory system performance and correctness.* ACM, 2006, pp. 82–91.

[47] J. Bloch, “Effective java (the java series),” 2008.

[48] G. J. Reuter and J. Gray, “Transaction processing: Concepts and techniques,” *MorganKaufmann, San Mateo, CA*, 1993.

[49] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[50] R. Kumar and K. Vidyasankar, “Hparstm: A hierarchy-based stm protocol for supporting nested parallelism,” in *the 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT’11)*, 2011.

[51] R. Guerraoui and M. Kapalka, “Opacity: A correctness condition for transactional memory,” Tech. Rep., 2007.

[52] P. Sestoft and H. I. Hansen, *C# precisely.* MIT Press, 2011.

[53] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint.* Elsevier, 2012.

[54] M. Mohamedin, B. Ravindran, and R. Palmieri, “Bytestm: Virtual machine-level java software transactional memory,” in *Coordination Models and Languages.* Springer, 2013, pp. 166–180.

[55] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.

[56] R. Ennals, "Software transactional memory should not be obstruction-free," Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Tech. Rep., 2006.

[57] S. Al Bahra, "Nonblocking algorithms and scalable multicore programming," *Queue*, vol. 11, no. 5, p. 40, 2013.

[58] S. M. Fernandes and J. Cachopo, "Lock-free and scalable multi-version software transactional memory," in *ACM SIGPLAN Notices*, vol. 46, no. 8. ACM, 2011, pp. 179–188.

[59] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "Mcrt-stm: a high performance software transactional memory system for a multi-core runtime," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming.* ACM, 2006, pp. 187–197.

[60] J. M. P. Cachopo, "Development of rich domain models with atomic actions," Ph.D. dissertation, Universidade Técnica de Lisboa, 2007.

[61] R. Zhang, Z. Budimlić, and W. N. Scherer III, "Commit phase in timestamp-based stm," in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures.* ACM, 2008, pp. 326–335.

[62] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski, "Anatomy of a scalable software transactional memory," in *Proc. 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[63] H. Avni and N. Shavit, "Maintaining consistent transactional states without a global clock," in *Structural Information and Communication Complexity.* Springer, 2008, pp. 131–140.

[64] T. Riegel, C. Fetzer, and P. Felber, "Time-based transactional memory with scalable time bases," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures.* ACM, 2007, pp. 221–228.

[65] R. Elmasri, *Fundamentals of database systems.* Addison-Wesley, 2011.

[66] R. C. Martin, *Clean code: a handbook of agile software craftsmanship.* Pearson Education, 2008.

[67] M. Fowler. (2015, May) Unit testing. [Online]. Available: http://martinfowler.com/bliki/UnitTest.html

[68] A. Nunes-Harwitt, “Cps recursive ascent parsing,” in *Proc. of Internat. LISP Conf*, 2003.

[69] D. Graham, E. Van Veenendaal, I. Evans, and R. Black, *Foundations of software testing: ISTQB certification.* Course Technology Cengage Learning, 2008.

[70] Microsoft. (2015, April) Compiler warning (level 1) cs0420. [Online]. Available: https://msdn.microsoft.com/en-us/library/4bw5ewxy(VS.80).aspx

[71] M. L. Scott, “Synchronization,” in *Encyclopedia of Parallel Computing.* Springer, 2011, pp. 1989–1996.

[72] Microsoft. (2015, May) Monitor class. [Online]. Available: https://msdn.microsoft.com/en-us/library/de0542zz.aspx

[73] ——. (2015, 2) Roslyn wiki. [Online]. Available: https://github.com/dotnet/roslyn/wiki

[74] ——. (2015, 2) Roslyn samples and walktroughs. [Online]. Available: https://github.com/dotnet/roslyn/wiki/Samples-and-Walkthroughs

[75] M. Torgersen. (2015, April) Languages features in c# 6 and vb 14. [Online]. Available: https://www.codeplex.com/Download?ProjectName=roslyn&DownloadId=930852

[76] M. Michaelis, “A c# 6.0 language preview,” *MSDN Magazine*, vol. May, pp. 16–23, 2014.

[77] ——, “The new and improved c# 6.0,” *MSDN Magazine*, vol. Oktober, pp. 18–24, 2014.

[78] M. Torgersen. (2015, April) Upcoming features in c#. [Online]. Available: https://www.codeplex.com/Download?ProjectName=roslyn&DownloadId=930852

[79] S. Imam and V. Sarkar, “Savina-an actor benchmark suite,” in *4th International Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE*, 2014.

[80] R. Guerraoui, M. Kapalka, and J. Vitek, “Stmbench7: a benchmark for software transactional memory,” Tech. Rep., 2006.

[81] C. A. R. Hoare, “Communicating sequential processes,” *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[82] J. A. Trono, “A new exercise in concurrency,” *ACM SIGCSE Bulletin*, vol. 26, no. 3, pp. 8–10, 1994.