Winter 12-15-2016

A Transactional Model and Platform for Designing and Implementing Reactive Systems

Justin Ross Wilson
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science
Department of Computer Science and Engineering

Dissertation Examination Committee:
Christopher Gill, Chair
Kunal Agrawal
Roger Chamberlain
Ron Cytron
Dave Peters
Gruia-Catalin Roman

A Transactional Model and Platform for Designing and Implementing Reactive Systems

by

Justin Wilson

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

December 2016
Saint Louis, Missouri

Contents

List of Tables

List of Figures

Acknowledgments

> Blessed be the God and Father of our Lord Jesus Christ, the Father of mercies and God of all comfort *2 Corinthians 1:3*

First and foremost, I would like to thank my wife who has been beside me every step of the way.

I am forever grateful for the patience, guidance, and wise counsel of my advisor Dr. Gill.

The seed for this dissertation was planted by Dr. Roman in the basement of Lopata Hall during a course on UNITY. He was lecturing on the development of structured programming and the give and take between formal methods and practice. I am grateful to Dr. Roman for introducing me to the clarity of thinking that comes from formal models.

I would like to thank the Department of Computer Science and Engineering, the National Science Foundation, and the Ford Motor Company for their generous financial support.

Justin Wilson

Washington University in Saint Louis
December 2016

ABSTRACT OF THE DISSERTATION

A Transactional Model and Platform for Designing and Implementing Reactive Systems

by

Justin Wilson

Doctor of Philosophy in Computer Science

Washington University in St. Louis, December 2016

Research Advisor: Professor Christopher Gill

A reactive program is one that has “ongoing interactions with its environment” [66]. Reactive programs include those for embedded systems, operating systems, network clients and servers, databases, and smart phone apps. Reactive programs are already a core part of our computational and physical infrastructure and will continue to proliferate within our society as new form factors, e.g. wireless sensors, and inexpensive (wireless) networking are applied to new problems.

Asynchronous concurrency is a fundamental characteristic of reactive systems that makes them difficult to develop. Threads are commonly used for implementing reactive systems, but they may magnify problems associated with asynchronous concurrency, as there is a gap between the semantics of thread-based computation and the semantics of reactive systems: reactive software developed with threads often has subtle timing bugs and tends to be brittle and non-reusable as a holistic understanding of the software becomes necessary to avoid concurrency hazards such as data races, deadlock, and livelock. Based on these problems

with the state of the art, we believe a new model for developing and implementing reactive systems is necessary.

This dissertation makes four contributions to the state of the art in reactive systems. First, we propose a formal yet practical model for (asynchronous) reactive systems called *reactive components*. A reactive component is a set of state variables and atomic transitions that can be composed with other reactive components to yield another reactive component. The transitions in a system of reactive components are executed by a scheduler. The reactive component model is based on concepts from temporal logic and models like UNITY [20] and I/O Automata [64]. The major contribution of the reactive component model is a formal method for *principled composition*, which ensures that 1) the result of composition is always another reactive component, for consistency of reasoning; 2) systems may be decomposed to an arbitrary degree and depth, to foster divide-and-conquer approaches when designing and re-use when implementing; 3) the behavior of a reactive component can be stated in terms of its interface, which is necessary for abstraction; and 4) properties of reactive components that are derived from transitions protected by encapsulation are preserved through composition and can never be violated, which permits assume-guarantee reasoning.

Second, we develop a prototypical programming language for reactive components called rc_{go} that is based on the syntax and semantics of the Go programming language. The semantics of the $\mathrm{rc}_{\mathrm{go}}$ language enforce various aspects of the reactive component model, e.g., the isolation of state between components and safety of concurrency properties, while permitting a number of useful programming techniques, e.g., reference and move semantics for efficient communication among reactive components. For tractability, we assume that each system contains a fixed set of components in a fixed configuration.

Third, we provide an interpreter for the $\mathrm{rc}_{\mathrm{go}}$ language to test the practicality of the assumptions upon which the reactive component model are founded. The interpreter contains an algorithm that checks for composition hazards like recursively defined transitions and non-deterministic transitions. Transitions are executed using a novel calling convention that can be implemented efficiently on existing architectures. The run-time system also contains two schedulers that use the results of composition analysis to execute non-interfering transitions concurrently. Fourth, we compare the performance of each scheduler in the interpreter to the performance of a custom compiled multi-threaded program, for two reactive systems. For one system, the combination of the implementation and hardware biases it toward an event-based solution, which was confirmed when the reactive component implementation outperformed the custom implementation due to reduced context switching. For the other system, the custom implementation is not prone to excessive context switches and outperformed the reactive component implementations. These results demonstrate that reactive components may be a viable alternative to threads in practice, but that additional work is necessary to generalize this claim.

Chapter 1

Introduction

1.1 Reactive Programs and Systems

Manna and Pnueli classify programs as either being *transformational* or *reactive* [66]. As implied by their name, transformational programs transform a finite input sequence into a finite output sequence. Transformational programs often can be divided into three distinct phases corresponding to the activities of input, processing, and output. A compiler is an example of a transformational program as it transforms a source file into an object file. Formal models of computation like the Turing machine [94] and λ-calculus [21] are concerned with transformational programs.

In contrast, a reactive program is characterized by "ongoing interactions with its environment" [66]. A reactive program must share resources with its environment to facilitate communication. From the perspective of the reactive program, input occurs when the environment acts on the reactive program and output occurs when the reactive program acts on the environment. The manipulation of resources that are internal to a reactive program is generically referred to as processing.

Whereas transformational programs are designed to halt and produce an output, reactive programs are often designed to run forever. The activities of input, processing, and output are thus recurring and often overlapped in the execution of a reactive program. Designers of reactive programs then must often reason about infinite event sequences containing interleaved input, processing, and output. A web server is a example of a reactive program as it repeatedly receives requests, processes them, and sends responses. Reactive systems include operating systems, databases, networked applications, interactive applications, and embedded systems. A number of formal models have been developed to reason about reactive systems, and include the Calculus of Communicating Systems [68], the Algebra of Communicating Processes [15], Cooperating Sequential Processes [32], Communicating Sequential Processes [52], Kahn Process Networks [56], the Actor Model [50][22][8], UNITY [20], and I/O Automata [64].

Asynchronous concurrency is a fundamental characteristic of reactive systems that makes them difficult to design and develop. Concurrency refers to the idea that a reactive program and its environment may act at the same time. In synchronous models, a reactive program and its environment share a common clock that allows the reactive program to coordinate access to shared resources, e.g., the environment writes a shared variable in one clock cycle and the reactive program reads it in the next. However, shared clocks are difficult to implement and do not scale. Consequently, many reactive systems are asynchronous, meaning that the reactive program and the environment evolve independently with respect to time. To facilitate communication, asynchronous models include facilities for atomicity that allow a reactive program to act without being interrupted by its environment and vice versa. These facilities for atomicity allow a reactive program to synchronize with its environment, as the atomic events play the role of a shared clock. A common approach to ensuring correctness in asynchronous models is to view the environment as an adversary that may deliver inputs at inopportune times.

State is fundamental to reactive systems and may be modeled directly or indirectly. Some examples of indirect approaches to modeling state include monads [96] which are typically used in functional languages such as Haskell, or mail queues with message-behavior pairs [8] which are used in actor-oriented languages such as Erlang. The imperative programming paradigm models state directly using variables and assignment and the dominant languages used to implement reactive systems, such as C, C++, and Java, are based on this paradigm. As we are interested in expressing reactive semantics directly, we limit the remaining discussion to reactive systems based on the imperative programming paradigm.

The imperative programming paradigm when presented using structured programming techniques attempts to express a (transformational) computation as a sequence of statements. The computation being performed is realized by executing each statement in the sequence where a statement is either an assignment statement, a condition (if/else), or a loop. Control flows from one statement to the next unless altered by a condition or loop. A locus of control is called a *thread*. Statements compose sequentially, that is, a sequence of statements can be thought of as a single statement that performs the computation of the sequence in one step. Imperative programming languages often permit the definition of procedures, e.g., subroutines, functions, macros, etc., as a way to abstract sequences of statements or expressions. Control passes from the calling sequence to the sequence indicated by the procedure and returns to the calling sequence when the procedure terminates. The semantics of calling a procedure are completely compatible with the sequential composition of statements.

Many reactive programs are multi-threaded and are designed to use the facilities of a conventional operating system. For the discussion to follow, we consider each thread in a multi-threaded program to be a reactive program. The environment for a thread then is the operating system and the other threads with which it directly interacts. This matches the definition of a reactive program as each thread will have ongoing interactions with its environment, i.e., interact with the operating system and other threads. This also matches the definition of asynchronous concurrency as the thread and its environment share no common clock and can interact asynchronously. To be more specific, a system call is asynchronous from the perspective of the operating system and threads may receive asynchronous signals from the operating system and from each other.

1.2 Trends

Developments in hardware and software platforms have resulted in an increasing demand for reactive systems. Embedded systems continue to proliferate due to advances in hardware that continue to produce new platforms, form factors, sensors, actuators, and price points, which allow embedded computers to be applied to a variety of application domains. Individuals, businesses, and governments are also deploying networks of sensors and actuators to monitor, control, and coordinate critical infrastructures such as power grids and telecommunication networks. These advances also have led to platforms for individual users, such as smart phones, e-readers, and tablets. Applications for these personal platforms are necessarily interactive and therefore reactive. The leveling-off of processor speeds and the resulting trend toward multi-core processors is also creating demand for reactive systems, as increases in performance must come through increasingly concurrent applications [89].

A general trend toward distribution is also driving demand for reactive systems. Increasingly, network services form the core (or are at least a critical component) of many applications and are fundamental to delivering the content (e.g., downloading books and movies) and communication (e.g., using social media) that drive the application. More and more devices are being equipped with (especially wireless) network adapters due to the introduction of inexpensive networking technologies. Networks are now also *emerging*, as opposed to being intentionally deployed, in environments such as the home, office, hospitals, etc. Applications that take advantage of these new networks are necessarily reactive. The trend toward distribution is already established in enterprise computing infrastructures where networked systems like file servers, print servers, web servers, application servers, and databases are critical or central to supporting business processes and achieving business objectives.

Given the continued proliferation of reactive systems, their number, diversity, and complexity is likely to increase as they encompass more and more interactions. This is most evident in large-scale distributed systems where a computation is spread over a variety of nodes. Such systems often *evolve* as new sub-systems are introduced and integrated into the existing infrastructure. The individual nodes themselves may also contain a variety of interactions. For example, it is not uncommon to find a smart phone application that concurrently interacts with the user via a graphical user interface, an application server via an Internet connection, and sensors (e.g., accelerometers) that are embedded in the hardware. Similarly, sophisticated servers like web servers and databases are often built from collections of interacting reactive modules.

1.3 Limitations of the State of the Art

Failure to account for asynchronous concurrency may result in a reactive program with *timing bugs*, meaning that correct execution depends on arbitrary scheduling decisions. Timing bugs may be manifested when the schedule of events is perturbed, e.g., due to the introduction of a new processor or operating system, or when part of the program takes more or less time than normal. Operating systems are particularly susceptible to timing bugs caused by device drivers [83]. Timing bugs can escape even a rigorous software development process and may lay dormant for years [62]. Furthermore, timing bugs are notoriously difficult to diagnose, inspiring the term *heisenbug*, a bug that disappears or changes its behavior when someone is attempting to find it (because the debugging process alters the timing of the program) [4]. Timing bugs can cost many hours of debugging time and can lead to a poor user experience, e.g., a non-responsive device or application, and loss of revenue, e.g., when an advertisement service cannot be used while a server reboots.

Introducing asynchronous concurrency into the imperative programming paradigm places a burden on developers as they must explicitly identify the statement sequences that must be atomic. The misuse of synchronization primitives, which becomes likely as state transitions become complex, may introduce concurrency hazards (e.g., deadlock [32]) which manifest themselves as timing bugs.

A number of design patterns for concurrency have been developed to help developers avoid concurrency hazards [84], [61]. These design patterns represent a move toward implicit atomicity as they often attempt to leverage language features to control atomicity (e.g., scoped locking [84]). While some of these patterns have been incorporated into programming languages (e.g., the `synchronized` keyword of Java implements the thread-safe interface pattern [84]), they are most often enforced only by convention and therefore easily violated or ignored (e.g., if a new developer is unaware of the convention). In practice, explicit atomicity and the corresponding use of synchronization primitives has proven to be tedious and error prone [89].

Correct synchronization in sequential imperative programs is a holistic problem that resists encapsulation. Sequential imperative programs, both transformational and reactive, are often designed using functionally modular principles such as procedural programming, object-oriented programming, and functional programming. A transition in a reactive program then is typically distributed over a variety of modules, i.e., the graph of procedure calls. The challenge for a developer then is to identify all of the shared state and then use synchronization primitives to guard concurrent updates. Proper synchronization is based upon a complete understanding of the call graph (which may not be fully known due to aliasing). The resulting code tends to be brittle as modifications tend to introduce new timing bugs.

1.4 Challenges

Two primary challenges must be addressed to overcome the current limitations noted in Section 1.3: reducing accidental complexity and providing techniques for composing and decomposing reactive systems in a principled manner. We discuss each of these challenges in turn.

Reduce accidental complexity. A key challenge towards adequately supporting complex reactive systems is to reduce the accidental complexity associated with their design and implementation. For software, accidental complexity is defined as the "difficulties that today attend its production but that are not inherent [18]." One source of accidental complexity is the *conflation of semantics*, where a problem naturally expressed using one set of semantics is implemented with a different set of semantics resulting in a semantic gap and obfuscation. To illustrate, Lee shows how the common practice of introducing thread-based concurrency via a library to an inherently sequential language significantly alters the semantics of the language [62]. As described in the previous section, we claim that the currently dominant approaches to developing reactive systems rely on inherently transformational languages that have been augmented with features for concurrency, which introduces an example of the kind of problem that Lee has identified. Thus, reducing the accidental complexity in reactive systems requires an approach that provides direct support for reactive semantics and addresses the inherent difficulties of asynchronous concurrency.

Achieve principled composition and decomposition. *Decomposition* and *composition* are essential techniques when designing, implementing, and understanding complex systems. Decomposition, dividing a complex system into a number of simpler systems, is often used when *designing* a system, e.g., through top-down design [100]. Composition, building a complex system from a number of simpler systems, is often used when *implementing* a system; i.e., simpler systems are implemented, tested, and integrated to create larger systems [18]. Often, the simplest systems in a design are common and can be reused across problem domains. Similarly, systems in the same problem domain often have common sub-systems. Thus, a common goal in software engineering is fostering design processes that produce and leverage reusable components. Decomposition also often imparts a logical organization to a system when the resulting sub-systems are cohesive, i.e., each sub-system has a well-defined purpose [75]. Thus, decomposition is a significant aid to understanding and managing complex systems.

Asynchronous concurrency undermines decomposition and composition when not properly encapsulated and therefore limits our ability to design and implement complex reactive systems. Such problems of asynchronous concurrency stem from the interactions between reactive programs. Decomposition increases the number of reactive programs constituting a system, which in turn increases the number of susceptible interactions and opportunities for timing bugs. For decomposition to achieve an overall reduction in complexity when designing a reactive system, it must reduce the amount of reasoning that must be performed at each level of the design. Thus, it must be possible to replace the details of how a reactive program is implemented with higher-level statements about its behavior in terms of its interface.

A second challenge then is to ensure that reactive systems can be decomposed and composed in a principled way. A design process based on composition and decomposition tends to be effective when the model adheres to certain principles:

1. The model should define units of composition and a means of composition. Obviously, a model that does not define a unit of composition and a means of composition cannot support a design process based on composition or decomposition.

2. The result of composition should either be a well-formed entity in the model or be undefined. Thus, it is impossible to create an entity whose behavior and properties go beyond the scope of the model and therefore cannot be understood in terms of the model. When the result of composition is defined, it should often (if not always) be a unit of composition. This principle facilitates reuse and permits decomposition to an arbitrary *degree*.

3. A unit of composition should be able to encapsulate other units of composition. When this principle is combined with the previous principle, the result is *recursive encapsulation* which permits decomposition to an arbitrary *depth*. Recursive encapsulation allows the system being designed to take on a hierarchical organization.

4. The behavior of a unit of composition should be encapsulated by its interface. Encapsulation allows one to hide implementation details and is necessary for abstraction.

5. Composition should be *compositional* meaning that the properties of a unit of composition can be stated in terms of the properties of its constituent units of composition. Thus, when attempting to understand an entity resulting from composition, one need only examine its constituent parts and their interactions. To illustrate, consider a system X that is a pipeline formed by composing a filter system F with reliable FIFO channel system C. This principle states that the properties of the composed Filter-Channel X can be expressed in terms of the properties of the Filter F and Channel C. Compositionality requires the ability to establish properties for units of composition that cannot be violated through subsequent composition.

6. Units of composition should have some notion of substitutional equivalence. If a unit of composition X contains a unit of composition Y, then a unit of composition X' formed by substituting the definition of Y into X should be equivalent to X. Substitutional equivalence guarantees that we can compose and decompose at will and summarizes the preceding principles. We believe that substitution should be linear in the size of the units of composition. To illustrate, suppose that X and Y are mathematical functions in the description above. If we take the size of a unit of composition to be the number of terms in the definition of a function then $|X'| \approx |X| + |Y|$.

A model adhering to these principles facilitates and supports *principled composition and decomposition*. Principled composition requires language support for interfaces, definitions, and substitutional equivalence. A variety of useful domains including mathematical expressions, object-oriented programming, functional programming, and digital logic circuits support principled composition.

Reactive programs based on threads are not necessarily subject to principled decomposition and composition. To illustrate, consider three imperative reactive programs (threads) A, X, and Y where A is composed of X and Y. Principled composition requires that the definition of A can be formed by substituting the definitions of X and Y. To preserve the reactive semantics when composing X and Y, one must consider all pairs of transitions, i.e., the Cartesian product, which represents all possible interleavings between the two statement sequences. The result of composition then is a two-dimensional torus where each node represents a compound state, each edge represents a transition, and each direction corresponds to executing a statement in X and/or Y. Appropriate measures must be taken to ensure that all transitions, i.e., all vertical, horizontal, and diagonal moves in the torus, are well-defined, i.e., explicit atomicity. We observe that 1) no existing platforms support the direct definition of such tori and 2) reasoning about a two-dimensional torus (or N-dimensional for N composed sequences) is qualitatively different that reasoning about a single sequence. Thus, reactive programs based on the imperative programming paradigm are not necessarily subject to recursive encapsulation and substitutional equivalence.

1.5 Approach and Contributions

To reduce the accidental complexity associated with the design and implementation of reactive systems while supporting principled composition and decomposition, we propose a transactional model for reactive systems called *reactive components*. A reactive component is a set of state variables and a set of atomic transitions that operate on those state variables. Linking a transition in one component to a transition in another component yields another transition that operates on the state variables of the constituent components. A transition-oriented approach resolves the main difficulties of the control-oriented imperative approach. In the reactive component model, transitions are atomic. This relieves developers from the burden of identifying and guarding critical sections. The composability of transitions allows developers to create complex state transitions independent of control flow. This relieves developers from needing a holistic understanding of the call graph when composing a complex state transition.

This research makes the following contributions to the state of the art in reactive system development. After presenting necessary background and related work in Chapter 2, we present the novel reactive component model in Chapter 3. In Chapter 4, we present $\mathrm{rc}_{\mathrm{go}}$: a programming language for reactive components based on the Go programming language. For tractability, we assume systems with a fixed number of components in a fixed configuration. Chapter 5 describes the implementation of an interpreter for $\mathrm{rc}_{\mathrm{go}}$ including the algorithm that checks a system for sound composition, a calling convention for transitions, the implementation of move semantics, and an approach to file descriptor I/O. Chapter 6 describes the design, implementation, and evaluation of two concurrent schedulers. The schedulers are compared to a custom multi-threaded application for two reactive systems. For one system, the reactive component implementation outperforms the custom application while the custom application outperforms the reactive component implementation for the other system. The results demonstrate that reactive components may be a viable alternative to threads but additional work is necessary to generalize this claim. Chapter 7 presents conclusions and describes future work that is motivated and enabled by this dissertation.

Chapter 2

Background and Related Work

In this chapter, we first present background on the semantics of reactive systems. We then provide a survey of other work related to this dissertation, with a particular emphasis on the UNITY and I/O Automata models upon which the approach presented here improves in specific ways.

Reactive semantics. State (memory) is fundamental to reactive systems as past inputs influence future behavior. Baeten concludes that the first step towards developing algebraic models for reactive systems was "abandoning the idea that a program is a transformation from input to output, replacing this by an approach where all intermediate states are important" [13]. The state of a reactive system is often captured in: program variables, e.g., in Dijkstra's Cooperating Sequential Processes [32]; messages in a channel or queue, e.g., in Milner's Calculus of Communicating Systems [68] and in the Actor Model [8]; or some combination of the two, e.g., in Kahn Process Networks [56].

Computation in a reactive system then can be viewed as a sequence of state transitions [77]. As these transitions may be complex, platforms typically allow complex state transitions to be composed from primitive state transitions and complex states, e.g., arrays, records, tuples, lists, sets, etc., to be composed from primitive states. Three orthogonal techniques for composing complex state transitions are expressions, sequential composition, and parallel composition. *Expressions* raise the level of abstraction by summarizing a computation whose intermediate results are unimportant. A compiler or interpreter is free to schedule the evaluation of an expression in any way that preserves the semantics of the expression.

Sequential composition is based on the idea that complex state transformations can be decomposed into a sequence of simpler state transformations. *Parallel composition* is based on the idea that complex state transformations can be composed by relating simpler state transformations, e.g., parallel assignment [14]. Conceptually, the right-hand side (RHS) of a parallel assignment is computed before any of the variables on the left-hand side (LHS) are modified.

We distinguish between a *reactive program* which is a static description of a set of transitions and a *reactive process* which is the realization of the transitions of the corresponding reactive program. State may be *private* meaning that it may only be updated by a single reactive process or *shared* meaning that it may be updated by multiple reactive processes. The stack associated with a thread and thread specific storage are common examples of private state. Shared state is often organized using abstraction where updates to shared state are exposed in the form of structured transitions as opposed to raw assignment, e.g., a method or function to place a message in a queue. Synchronization and communication are identified as necessary activities in a reactive system [9]. The two approaches to communication are *shared variables* and *message passing* [9].

Multiple reactive processes effect state transitions that overlap in time, which is called *concurrency*. Simultaneous state transitions, i.e., those that overlap in real time, require parallel physical resources. State transitions that are formed by sequential composition may be overlapped by interleaving the primitive transitions of the corresponding complex transitions. The result of concurrent state transitions that update the same (shared) state may be undefined. Consequently, updates to shared state must be coordinated to prevent corruption.

Platforms for reactive systems, therefore, include the notion of *atomicity* which says that certain transitions may not be interrupted, i.e., executed simultaneously or interleaved with another transition. An *event* is an atomic state transition.

Non-determinism is another inherent attribute of reactive systems that conveys the idea that the order of events in a reactive system is not fixed. Non-determinism is typically combined with atomicity to ensure that transitions are well-defined. In a pair of events operating on the same state, for example, atomicity says that one event will be executed before the other while non-determinism says that the order in which they are executed is not determined.

True concurrency, i.e., simultaneity, is often modeled using a non-deterministic sequence of atomic events, e.g., [64], [20], [66].

As a sequence of atomic transitions, a reactive process (or rather the reactive program that defines it) may either have *deterministic sequencing* or *non-deterministic sequencing.* As implied by the name, the order of transitions in a reactive process with deterministic sequencing is completely determined, i.e., there is always a single next transition (or termination). The sequence of state transitions is called a *flow of control.* Reactive processes with deterministic sequencing are often based on an (infinite) loop that repeats for the duration of the reactive process. Influential models based on deterministic sequencing include Dijkstra's Cooperating Sequential Processes [32] and Hoare's Communicating Sequential Processes [52].

Conversely, the order of transitions in a reactive process with non-deterministic sequencing is not completely determined, i.e., the next transition is selected from a set of candidates. Platforms supporting reactive processes with non-deterministic sequencing include a *scheduler*, which chooses among the available transitions. Reactive programs with deterministic sequencing correspond to a (circular) list of transitions while reactive programs with non-deterministic sequencing correspond to a set of transitions. Deterministic sequencing and non-deterministic sequencing only describe individual reactive processes as the global choice for the next transition is in general non-deterministic. Influential models based on non-deterministic sequencing include the UNITY model of Chandy and Misra [20] and Lynch's I/O Automata [64].

Atomicity may either be *explicit* or *implicit* in a model for reactive systems. Platforms that support reactive processes with deterministic sequencing and shared variables typically include primitive transitions called *synchronization primitives*, e.g., test-and-set, compare-and-swap, that may atomically update state and/or alter the flow of control [9]. Synchronization primitives can be used to construct more general synchronization mechanisms like semaphores and monitors [32]. The goal of synchronization is to create atomic sequences of transitions called *critical regions* or *critical sections* [9]. Atomicity, therefore, is made explicit by the programmer. Message passing combines communication with synchronization to achieve implicit atomicity in reactive programs based on deterministic sequencing [9]. Non-deterministic sequencing requires that all transitions be atomic and therefore implies implicit atomicity.

Related work. Reactive systems are designed and implemented using shared variables and/or message passing. A popular approach to reactive systems is the pairing of shared variables with deterministic sequencing and explicit atomicity as is done in Dijkstra's Cooperating Sequential Processes [32]. Andrews and Schneider describe a number of techniques associated with this approach including coroutines, fork/join, spin locks, semaphores, conditional critical regions, and monitors [9]. This model is supported by widely available platforms, i.e., operating systems, via processes with shared memory or threads [86]. A number of architectural patterns have been developed based on this approach, e.g., [84], [61]. As described by Lee [62], support for this model can be integrated into an existing sequential language through an external library, e.g., POSIX threads, or extensions to the base language, e.g., Cilk [16], Split-C [28], C++11 [5]. Sutter and Larus [89] and Lee [62] provide a modern perspective on the difficulties associated with this approach. In [89], the authors call for "OO for concurrency–higher-level abstractions that help build concurrent programs, just as object-oriented abstractions help build large componentized programs." The work presented in this dissertation is a step in this direction.

Transactional memory has been proposed as an alternative to locks when synchronizing multiple processes. Transactional memory was inspired by the atomic transactions of databases [36]. Knight [57] and then Herlihy and Moss [49] proposed cache-based hardware support for transactional memory. Transactional memory is forthcoming on modern processors [81]. Software transactional memory, proposed by Shavit and Touitou [85], has sparked a great deal of interest and has been implemented in a number of languages, e.g., Clojure [48].

Another technique for designing and implementing reactive systems that is receiving renewed interest is promises and futures [40, 72]. Promises and futures represent two sides of a deferred computation. The consumer of a deferred computation receives a future that it can later interrogate for the values produced by the deferred computation. The producer of a deferred computation receives a promise that it later fulfills by performing the deferred computation.

Another popular approach to reactive systems is messaging passing with deterministic sequencing as is done in Hoare's Communicating Sequential Processes [52]. This model is also supported by widely available platforms via pipes, message queues, and sockets [86]. Go is a modern programming language with language support for the Communicating Sequential Processes model [2]. A message passing channel may either be unbounded or have a fixed size and may be accessed synchronously or asynchronously by either the sender or the receiver [9].

Events, as proposed by Ousterhout [73], and embodied in the Reactor and Proactor architectural patterns [84], offer a popular technique for structuring user applications based on deterministic sequencing. The application is designed around a loop that multiplexes I/O events from the operating system using a polling function like `select` or `poll`. In response to an I/O event, the application invokes an (atomic) event handler that may update the state of the process and perform non-blocking I/O on various channels. Events invert the flow of control since high-level functions, e.g., processing a message, are triggered by low-level functions, e.g., receiving a byte. The context of each computation must be managed explicitly which is referred to as "stack ripping" [7]. Event handlers from different logical computations may be interleaved giving the illusion of concurrency while avoiding the challenges of synchronization. Event systems wishing to take advantage of true concurrency must use multiple event loops and face all of the challenges of multi-threaded programming. Node.js [88] and ECMA Script (JavaScript) [34] are two modern programming languages based on an event loop.

UNITY and I/O Automata. The UNITY model of Chandy and Misra [20] and the I/O Automata model of Lynch [64] are two influential models for reactive systems based on non-deterministic sequencing. In these models, a scheduler repeatedly executes transitions selected non-deterministically from the set of possible transitions. The scheduler is assumed to be fair, meaning that it will execute a transition an infinite number of times in an infinite execution. Transitions correspond to *(parallel) assignment statements* in the UNITY model and *actions* in the I/O Automata model. The UNITY model contains two means of composing programs which suggests that a model based on non-deterministic sequencing could support principled decomposition. Creating a program via the *union* operation involves taking the union of the state variables (name-based equivalence) and assignment statements of the constituent programs. *Superposition* is a means of composition that transforms an underlying program into another. The transformation is allowed to add new state variables, add new assignment statements with the limitation that they only update new variables, and augment existing assignment statements but only by adding clauses that modify new variables. The size of the resulting program (measured in assignment statements) is on the order of the sum (as opposed to the product) of the sizes of the two constituent programs.

Superposition is property-preserving while union is not property-preserving. However, superposition has certain weaknesses, as described by the authors of UNITY [20]:

> Both union and superposition are methods for structuring programs. The union operation applies to two programs to yield a composite program. Unlike union, a transformed program resulting from superposition cannot be described in terms of two component programs, one of which is the underlying program. The absence of such a decomposition limits the algebraic treatment of superposition. Furthermore, a description of augmentation seems to require intimate knowledge of statements in the underlying program. Appropriate syntactic mechanisms should be developed to solve some of these problems.

They go on to note that a restricted form of superposition, i.e., one that only adds variables and assignment statements, is equivalent to union and can be analyzed as such.

The essence of superposition is that a transition in one program can be linked to a transition in another program, i.e., parallel composition. Whereas UNITY lacks language support for doing so, the I/O Automata [64] model presents a partial solution by associating names with transitions (actions). Actions in different Automata can then be composed on the basis of name matching. The I/O Automata model defines three kinds of actions: output actions, input actions, and internal actions. Output actions and internal actions, collectively called local actions, contain guards and may be executed by the scheduler. Each input action must be composed with an output action to be executed. Output actions can produce values that are consumed by input actions. Since the state of each I/O automaton is private, composition in the I/O Automata model is property-preserving.

Of interest to this dissertation is the improvement and implementation of these ideas and their application to the design and development of reactive systems. Granicz et al. propose a compilation method for UNITY in their Mojave compiler framework [46]. Other initiatives to implement the UNITY programming language are summarized in [46]:

Few compilers have been developed for the UNITY language. DeRoure's parallel implementation of UNITY [31] compiles UNITY to a common backend language, BSP-occam; Huber's MasPar UNITY [54] compiles UNITY to MPL for execution on MasPar SIMD computers; and Radha and Muthukrishnan have developed a portable implementation of UNITY for Von Neumann machines [79][1].

Goldman's Spectrum Simulation System [44] allows one to simulate systems expressed as I/O Automata. The IOA toolkit is an implementation of I/O Automata focused on verification and simulation [3]. The IOA toolkit does contain a source-to-source compiler (IOA to Java) and a run-time system that has been used to compile and execute distributed protocols [43].

Summary. Formal models of reactive systems may be organized around sequential threads with shared variables, e.g., Cooperating Sequential Processes [32], sequential threads with message-passing, e.g., Communicating Sequential Processes [52], atomic transitions with shared variables, e.g., UNITY [20], and atomic transitions with message-passing, e.g., I/O Automata [64]. Models based on atomic transitions avoid reasoning about the interleaved execution of threads which may simplify reasoning about reactive systems. Combining atomic transitions with message passing (I/O Automata) creates the opportunity for property-preserving composition which is problematic in models based on shared variables, e.g., UNITY. The reactive component model described in Chapter 3 of this dissertation extends the property-preserving composition techniques of I/O Automata by permitting composition to an arbitrary depth and degree using the superposition technique of UNITY to link transitions in different modules in a principled way.

Reactive semantics may be introduced via libraries or built into a language. Taking an inherently transformational language and introducing reactive semantics via a library may significantly alter the semantics of the language. With respect to language support, an inherently transformational language may be augmented with features to support reactive semantics or the language can be designed around a particular approach to reactive systems. Go with CSP-style primitives is an example of the former and Erlang as a realization of the Actor model is an example of the latter. Language support has the potential to raise the level of abstraction and provides the opportunity to check and enforce reactive semantics. To

[1]We believe all of these projects, including [46], are now defunct.

date, the main focus areas of languages designed around the atomic transition paradigm have been parallel computing and simulation. The rc$_{\text{go}}$ language presented in Chapter 4 of this dissertation instead focuses on practical concerns of software engineering such a reference semantics for the efficient implementation of data structures, move semantics for efficient communication, and the isolation of state for property-preserving composition. Similarly, there are few results concerning the design, implementation, and evaluation of fair schedulers, which are a necessary piece of run-time systems that support atomic transitions. Chapter 5 of this dissertation describes the design, implementation, and evaluation of two concurrent fair schedulers.

Chapter 3

Reactive Component Model

In this chapter, we present a new model for composing and decomposing reactive programs via *reactive components*. The model is biased toward practical software development even as it enforces properties based in formal methods. Consequently, the model favors utility, practicality, flexibility, and ease of implementation. Unlike UNITY in which composition is not property-preserving, the composition of reactive components is property-preserving which facilitates hierarchical and modular reasoning. Unlike I/O Automata in which transitions have limited depth, a transition among reactive components may access and cascade to an arbitrary number of other components, which permits decomposition to an arbitrary depth and degree.

3.1 Features of the Model

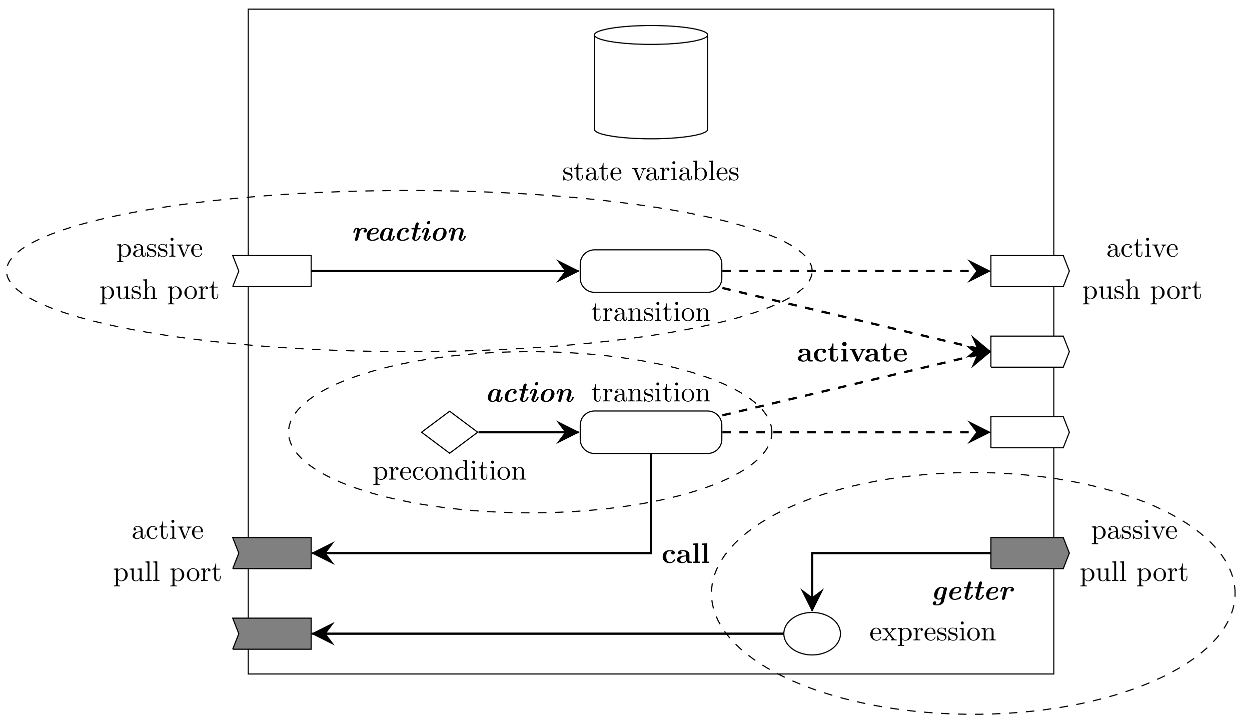

Figure 3.1: Features of a reactive component

Figure 3.1 shows the major features of a reactive component. As in other state-based formal models like UNITY [20] and I/O Automata [64], the core of a reactive component in this model is a set of *state variables* and a set of atomic *transitions* that manipulate those state variables. When reasoning about a system, behavior is expressed as propositions over the state variables where the propositions are derived from the transitions.

The reactive component model defines interface elements and composition semantics that allow reactive programs to be composed in a principled way. For example, an *active push port* allows a transition in one component to be linked to a transition in another component such that the resulting combined transition is atomic. Active push ports allow reactive components to publicize their behavior. An active push port may be *bound* to and conditionally *activate* zero or more *passive push ports*. A passive push port names the corresponding transition that will be executed when the passive push port is activated. This combination of passive push port and transition is called a *reaction* because it reacts to a transition in another component.

The atomic linkage of a transition in one component to a transition in another component through the push port mechanism allows the properties of each component to be related to one another and more complex systems to be constructed by composing simpler systems. Transitions that are not executed via a passive push port are executed by the scheduler. A transition of this kind may be governed by a Boolean expression called a *precondition*. The combination of a precondition and transition is called an *action* because it is a voluntary transition under the control of the containing component.

An *active pull port* represents an immutable external data dependency. The component may *call* an active pull port to yield a value required in a transition. Active pull ports must be bound to a *passive pull port* which resembles a function returning one or more values. Associated with a passive pull port is an expression that may interrogate the state of the corresponding component or call other pull ports. The combination of a passive pull port and an expression is called a *getter*. Where push ports allow components to publicize their behavior, pull ports allow components to safely publicize their internal state for use by other components.

Composition in the reactive component model has two main features. The first is recursive encapsulation where a state variable in one component may represent an instance of another reactive component. The second is the ability to bind push ports and pull ports through an *explicit* set of *bindings*. The decision to use an explicit set of bindings (as opposed to implicit named-based matching) is more in keeping with the goals and techniques of practical software development, since it facilitates the use of software developed under different naming conventions, i.e., third-party software. A third minor feature called *exporting* is the ability of an encapsulating component to adopt interface elements of its sub-components without defining complimentary ports and transitions that do nothing but forward an activation or call. The ability to publicize behavior and state and the ability to assemble well-defined behavior from existing behaviors in a straight-forward and flexible way are thus defining characteristics of the reactive component model.

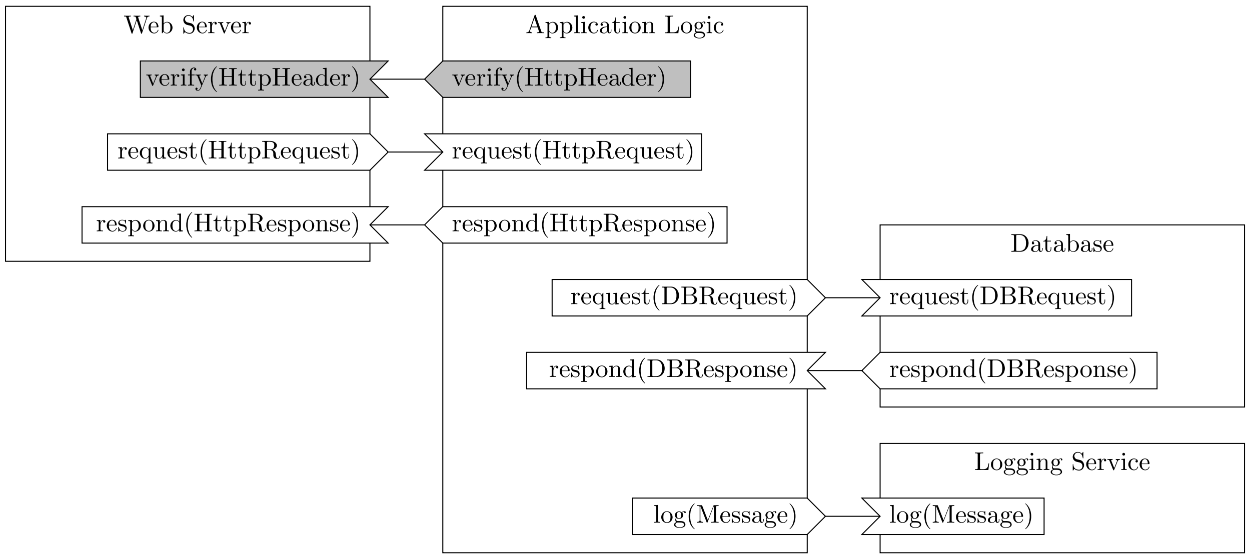

Figure 3.2: Diagram of a web application built using reactive components

To demonstrate the utility of component interfaces and explicit support for property-preserving composition, we now present an illustrative design for a web application using reactive components, as is shown in Figure 3.2. The web application consists of five reactive components. The first is an unnamed top-level component representing the complete application. This component has four sub-components representing a web server, the application logic, a database, and a logging service. The ports of the application logic component have been bound to the corresponding ports in the web server, database, and logging service components. The interface of a component, which consists of its ports, provides insight into the behavior of the component. For example, based on the interface of the web server component we may expect it to 1) verify incoming HTTP requests[2], 2) pass on valid HTTP requests to the application, and 3) accept HTTP responses from the application.

Figure 3.2 also demonstrates how reactive programs can be constructed by composing reactive components, so that a developer may focus on the application logic and use existing components for the web server, database, and logging service. Furthermore, one can imagine developing three stateless components surrounding the application logic that do nothing but translate messages between application specific message types and the generic types required by the web server, database, and logging service. The application logic, then, is completely

[2]A production web server may verify that the requested HTTP method can be applied to the given URI, that content length limits are respected, that content types are supported, etc.

isolated from the surrounding libraries and may be tested by providing mock components for the web server, database, and logging service. The application logic itself has a well-defined interface and could be reused, say, by implementing a graphical front-end to drive the application instead of a web service.

3.1.1 State Variables

Part of the internal core of a reactive component is a set of state variables, which are the subject of the propositions that demonstrate the behavior of the system. The state variables are manipulated by the assignment statements that constitute the transitions. As in other formal models, the types of the state variables may be selected to make writing proofs easier. However, implementations of the reactive component model must provide a concrete type system that allows the state variables to be realized on a given machine. For example, the $\mathrm{rc}_{\mathrm{go}}$ language for reactive components presented in Chapter 4 of this dissertation uses the type system of Go to define the types of state variables. The type of a state variable also may be a reactive component type, which facilitates recursive encapsulation. For modeling purposes, state variables typically have *value semantics* to avoid reasoning about references and aliasing. However, and as a practical concession, we will introduce pointers for building arbitrary linked data structures (since we advocate an imperative state-based implementation) and discuss issues raised by introducing pointers into the reactive component model in Chapter 4.

3.1.2 Atomic State Transitions

The rest of the internal core of a reactive component is a set of atomic state transitions that manipulate the state variables. A precise definition of the atomicity of state transitions will be deferred to the subsequent sections on composition (Section 3.1.3) and execution (Section 3.1.5). State variables and state transitions are private, meaning that transitions can only refer to the state variables of the associated reactive component and a transition in one reactive component can only be linked to a transition in another component through the composition mechanisms. Thus, all initialization and updates to state variables can be determined by examining the transitions of the corresponding component. The encapsulation of state variables contribute to composition being *compositional*. All properties established from the transitions of a component will continue to hold as the component participates in composition since the set of transitions that affect the state variables is fixed. State transitions must be deterministic meaning that the next value of every state variable is uniquely and well defined. State transitions defined by a sequence of simple assignments are implicitly deterministic. However, special care must be take to ensure that state transitions are deterministic when described using parallel assignment statements as the same variable may appear on the left-hand-side and be assigned two different values [20]. As described in Sections 3.1.3 and 3.1.5, composition links transitions using parallel assignment which creates an opportunity for non-deterministic state transitions. The problem of non-deterministic state transitions arising from composition is explored in Section 3.3.

The reactive component model presented in this chapter does not prescribe a specific language for encoding state transitions. The language used depends on the goals and tastes of the one developing or analyzing the model. For example, expressing transitions using the programming language of UNITY [20] may allow the modeler to extract proofs from the text, a major goal of UNITY. The approach used by I/O Automata [64] is to specify the condition established by each transition. As with state variables, we present a language that uses the statements and expressions of the Go programming language to encode state transitions in Chapter 4. This furthers our goal of making reactive components approachable by a general software engineering audience.

3.1.3 Actions, Reactions, Push Ports, Bindings, and Composition

A state transition is either part of an action or a reaction. An *action* is a state transition whose execution is under the control of the component to which it belongs. The action is guarded by a *precondition* that is guaranteed to be true the instant before the action is executed. The precondition is a Boolean expression that determines if the action is *enabled* or *disabled*. If the precondition is absent, it is assumed to be true.

A *reaction* is a state transition whose execution is under the control of another action or reaction. Thus, we require a mechanism for linking a reaction to an action or another reaction. To do this, we introduce the notion of a *push port*. A push port is a typed interaction point consisting of an active side that conditionally *activates* the port and provides arguments to the port, and a passive side that *reacts* to the port and may access the arguments provided by the active side. A reaction, therefore, is the combination of a passive push port and a state transition.

A *binding* is a declaration that associates an active push port in one component with a reaction in another component. The transition associated with the reaction is executed atomically with the transition that activates the active push port. Composition in the reactive component model is achieved by declaring sub-components (recursive encapsulation) and linking transitions via binding.

3.1.4 Transactions

A *transaction* is a compound and concrete state transition formed by expanding the activations in an instance/action pair, subject to the bindings in a system. The instance and action that define a transaction are called the *root* of the transaction. A transaction is enabled/disabled if its root action is enabled/disabled. Given the root instance and action, we may enumerate the active push ports that may be activated by the transition of the action. Note that the activation of a push port is conditional, being under the control of the transition. The push ports, in turn, activate reactions in particular component instances. The transitions associated with the reactions may activate other push ports and so on. This analysis can be repeated to discover all of the reactions that are linked to the root action. The result is a *transaction graph* which is a directed acyclic graph $G = (N, E)$. Each node $n \in N$ is a pair (i, x) where i is a component instance and x is either an action, reaction, activation, or push port. Each edge $e \in E$ corresponds to a causal relationship between an action/reaction and an activation, an activation and a push port, or a push port and a reaction. The edges between actions/reactions and activations indicate that the action/reaction *may* execute the activation, as they may be conditionally executed. The edges between activations and push ports and between push ports and reactions indicate that the implied push port or reaction *will* be executed if the upstream activation is executed.

3.1.5 Execution

We adopt the common practice of modeling concurrency with non-deterministically executed atomic actions as is done in UNITY [20], I/O Automata [64], and the Actor Model [8]. The execution of transactions is performed by a *scheduler*. The scheduler executes one transaction at a time, thus, each transaction is *atomic* with respect to all other transactions. A transaction is *enabled* if its precondition is true. When executing a transaction, the scheduler first evaluates the precondition and then executes the body of the transaction if the transaction is enabled. Thus, executing an enabled transaction *may* result in a state change while executing a disabled transaction *never* causes a state change. The scheduler is *fair* meaning that each transaction is executed an infinite number of times. The system may reach a *fixed point* where all transactions are disabled. The order in which transactions are executed is not determined, thus, execution is *non-deterministic*.

We divide a state transition into two phases called the *immutable phase* and the *mutable phase*. Logically, a transition assigns values to a set of state variables. This can be modeled as a parallel assignment statement with a left-hand side (LHS) consisting of a list of state variables and a right-hand side (RHS) consisting of a list of expressions that provide the next value for each corresponding state variable. The immutable phase corresponds to the computation of the RHS in a state transition. The mutable phase corresponds to the update of the values on the LHS with the values on the RHS. When transitions are linked with composition, we must relate the mutable and immutable phase in one state transition to the mutable and immutable phases of the other transitions in the transaction. Let A be a transition (action) and R be a transition (reaction) that is activated by A. Let A_I be the immutable phase of A, A_M be the mutable phase of A, R_I be the immutable phase of R, and R_M be the mutable phase of R. For the sake of argument, assume that A_I reads variables that are written in R_M and that R_I reads variables that are written in A_M. We require that A_I be evaluated before A_M, R_I be evaluated before R_M, and A_I be evaluated before R_I since activation is conditional. This leaves three possible sequences:

- $A_I A_M R_I R_M$. In this sequence, a variable is first updated in A_M and then the updated value is read in R_I. The issue with this interpretation is that it does not compose well. Ideally, we would like to be able to rewrite the transition as a single transition consisting of a single immutable phase and mutable phase.

- $A_I R_I A_M R_M$ and $A_I R_I R_M A_M$. These sequences resolve the issue with the first sequence by providing a clear immutable phase ($A_I R_I$) and mutable phase ($A_M R_M$ and $R_M A_M$).

Thus, with respect to transactions, all immutable phases (which includes all push port activations) are performed before all mutable phases.

3.2 Example: Clock System

To illustrate the behavior of reactive components under this model, we rewrite the *Clock automaton* example in [64] using reactive components. The Clock automaton consists of a free-running counter and flag used to implement a request-response protocol. Our Clock component is defined in Figure 3.3. The state variables are identified with `var` and their initial values are provided. The component contains a reaction named `request` for receiving requests to sample the current value of the counter. The component also contains an active push port named `clock` for communicating the sampled value of the counter. The `Clock` action is conditioned on the flag variable (which indicates that a request has been made) which when it executes resets the flag and communicates the value of `counter` by activating the `clock` port. The `Tick` action increments the free-running counter. The absence of a precondition means this action is always enabled. A simple client for the Clock component that perpetually requests the current count is shown in Figure 3.4.

In isolation, a Clock component will increment its counter forever and a Client component will make a request and then stop. In order to make the two components work together, we must compose them. Figure 3.5 shows a System component that instantiates a Clock component and a Client component and binds the corresponding push ports in each instance. In the composed system, the client's `Request` action will be executed activating the `request` port which in turn causes the `request` reaction in the clock to be executed. Figure 3.6 shows the transaction graph for the `Request` action. Eventually, the clock's `Response` action will be executed activating the `clock` port which in turn causes the `clock` reaction in the client to be executed with the current value of the clock's counter. Figure 3.7 shows the transaction graph

```
component Clock {
  var int counter (0)
  var bool flag (false)
  push clock(int t)

  reaction request() flag := true

  Clock: flag -> flag := false activates clock(counter)

  Tick: counter := counter + 1
}
```

Figure 3.3: Definition of the Clock component of the Clock System

```
component Client {
  var bool flag (false)
  push request()

  Request: !flag -> flag := true activates request()

  reaction clock(int t) flag := false || /* do something with t */
}
```

Figure 3.4: Definition of the Client of the Clock System

```
component System {
  var Clock clock
  var Client client

  bind {
    client.request -> clock.request
    clock.clock -> client.clock
  }
}
```

Figure 3.5: Definition of the System component of the Clock System

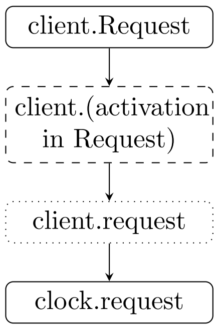

Figure 3.6: Transaction diagram for the `client.Request` action of the Clock System

for the `Clock` action. In between these actions, the `Tick` action of the clock is incrementing the counter.

3.3 Properties of Composition

In this section, we examine various features related to reactive components and composition. Substitutional equivalence for reactive components is demonstrated by outlining a procedure for in-lining sub-components. Hazards of composition, namely, non-deterministic state transitions resulting from conflicting and recursive composition, are identified and a means of detecting them is proposed. The issue of decomposition is considered and *pull ports* are introduced as a mechanism for decomposition.

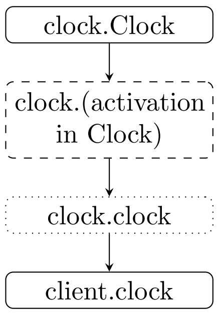

Figure 3.7: Transaction diagram for the `clock.Clock` action of the Clock System

```
component System {
  /* Substitution of clock component. */
  var int clock_counter (0)
  var bool clock_flag (false)
  push clock_clock(int t)

  reaction clock_request() clock_flag := true

  clock_Clock: clock_flag -> clock_flag := false activates clock_clock(clock_counter)

  clock_Tick: clock_counter := clock_counter + 1

  /* Substitution of client component. */
  var bool client_flag (false)
  push client_request()

  client_Request: !client_flag -> client_flag := true activates client_request()

  reaction client_clock(int t) client_flag := false || /* do something with t */

  bind {
    client_request -> clock_request
    clock_clock -> client_clock
  }
}
```

Figure 3.8: Substitution of state variables, ports, actions, and reactions for the sub-components of the System component of the Clock System

```
component System {
  var int clock_counter (0)
  var bool clock_flag (false)
  var bool client_flag (false)
  push clock_clock(int t)
  push client_request()

  clock_Clock: clock_flag -> clock_flag, client_flag :=
    false, false activates clock_clock(clock_counter) ||
    /* do something with clock_counter */

  client_Request: !client_flag -> client_flag, clock_flag :=
    true, true activates client_request()

  clock_Tick: clock_counter := clock_counter + 1
}
```

Figure 3.9: Simplifications of ports, bindings, and transitions in the expanded System component of the Clock System

3.3.1 Substitutional Equivalence

For reactive components, substitutional equivalence means that a sub-component can be replaced with its definition and the result is a well-defined entity in the model. To this end, a procedure for substituting the definition of a sub-component involves 1) renaming and adding all state variables, ports, actions, and reactions to the parent component and 2) simplifying bindings by substituting the transitions associated with a reaction into the action or reaction that activates the reaction in question.

To illustrate, Figure 3.8 shows the result of substituting state variables, ports, actions, and reactions into the System component of Figure 3.5. Identifiers in the sub-components have been prefixed with the name of the sub-component instance to avoid name clashes. For example, the `request` reaction in the `clock` sub-component has been renamed to `clock_request`. Figure 3.9 shows the result of simplifying bindings and state transitions. Note that the push ports have been retained for subsequent composition, i.e., the System component may be a component in a larger system. The result is a reactive component whose "size" in terms of state variables, actions, and reactions is the sum of the sizes of its constituent components.

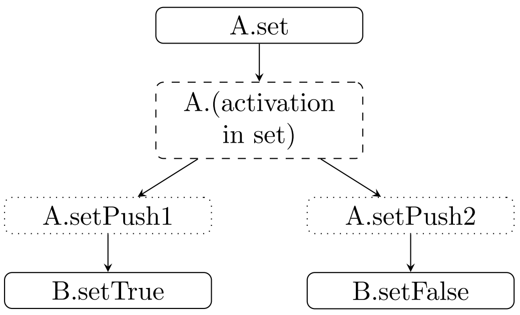

Figure 3.10: Transaction diagram for a non-deterministic transaction

Substituting the definition of the Clock component and Client components into the System component confirms the intuition that the flag variable in the client and flag variable in the clock are the same since 1) initially they have the same value and 2) they take on the same value in every state transition.

3.3.2 Determinism and Composition

The result of composing two well-defined reactive components may yield a system that is non-deterministic. The two "hazards" that must be avoided are non-deterministic assignment to state variables and recursively activated reactions. Non-deterministic assignment results when the next value for a state variable is not well defined due to the inclusion of two or more transitions in a transaction that operate on the state variable. To illustrate, consider the transaction depicted in Figure 3.10. The set action of component A has a single activation that activates both the setPush1 and setPush2 push ports. The setPush1 port is bound to the setTrue reaction of component B while the setPush2 port is bound to the setFalse reaction of the same component B. Suppose that the setTrue reaction sets a flag to true while the setFalse reaction sets the same flag to false. The transaction is non-deterministic because the value of the flag after the (A,set) transaction may either be true or false. An offending pair of transitions may appear anywhere in a transaction graph given that arbitrary transaction graphs may be constructed through composition. If the underlying language used to define transitions admits pointers, dynamic memory, if statements, and loops (i.e., is Turing-complete), then the problem of determining if two transitions operate on the same state variable is undecidable in general [59, 80]. For a transaction graph G, instances i_1

and i_2, and actions/reactions t_1 and t_2, a necessary (but not sufficient) condition for non-deterministic assignment NDA from composition is $NDA(G) : \exists (i_1, t_1), (i_2, t_2) \in G.N, i_1 = i_2, t_1 \neq t_2$ which says that the transaction must contain two different transitions involving the same component instance.

A recursively activated transition occurs when the transaction graph has a cycle. The execution of such a transaction may result in well-defined next values for all state variables assuming that 1) the recursion is bounded and 2) the parameters passed to every reaction result in identical computations. A transaction may be analyzed like a traditional transformational program since it has finite input, a finite output, and should terminate. The first problem, then, is a thinly disguised version of the halting problem since it asks if a computation (the transaction) expressed in a Turing-complete language terminates (has bounded recursion) [95, 30]. A bounded recursion means that the execution of the transaction will generate a bounded number of activations A. For each activation $a \in A$, we must determine what state is updated by a. If the language is Turing-complete, then this problem is undecidable in general [59, 80]. For state variables that are updated by more than one activation, we must then show that each activation sets the state variable to the exact same next state. If we treat each activation as a program, then we require a function that determines if two (arbitrary) programs compute the same function, which is again, undecidable in general [82].

The difficulty of detecting composition that results in non-deterministic assignments suggests that these problems are best checked by a machine. That is, an implementation of reactive components may prevent non-deterministic assignment by checking that $NDA(G)$ is false for all transaction graphs in the system. Similarly, an implementation may check for recursive activation by checking for cycles in transaction graphs. Both of these approaches are used by the implementation described in Section 5.2.

3.3.3 Decomposition, Getters, and Pull Ports

Substitutional equivalence implies that the process of substituting the definition of a sub-component into a parent component may be reversed and that sub-components may be "factored out" of an existing component, e.g., for reuse with other components.

Another motivation for decomposition is potentially increased performance through parallelism. Recall that true concurrency in reactive components is modeled as the serial and non-deterministic execution of atomic actions. Two transactions can be safely executed concurrently if it can be shown that the state variables involved in each transaction are disjoint. As was previously mentioned, making this determination is undecidable for Turing-complete languages in the general case. However, the problem becomes decidable if the component instance is used as a proxy for its constituent state variables. Let $rw : t \rightarrow \{Read, Write\}$ be a function that maps a transition to a value indicating that variables are read-only during the transition or written in some way. Two instance/transition pairs are independent (*indp*) if either the instances are different or at most one of the transitions writes to the variables of the instance:

$$indp((i_1, t_1), (i_2, t_2)) : i_1 \neq i_2 \vee \neg(rw(t_1) = Write \wedge rw(t_2) = Write) \tag{3.1}$$

Two transaction graphs are independent if all of their nodes are independent $indp(G_1, G_2) = \forall (i_1, t_1) \in G_1.N, (i_2, t_2) \in G_2.N \;\; indp((i_1, t_1), (i_2, t_2))$. The significance of the preceding analysis is that the determination about what actions can be executed concurrently becomes machine checkable due to the strong guarantee that state variables belonging to an instance can only be modified by the transitions of that instance.

To illustrate the mechanisms required for decomposition, we will factor out a Counter component from the Clock component of Figure 3.3 and then rewrite the Clock component using the Counter component. Upon inspection, the `Tick` action and `request` reaction can be executed concurrently since the state variables involved in each transition are disjoint. Figure 3.11 shows the Counter component which consists of a `counter` state variable. The `Tick` action can be moved to the Counter component without complication. The `Clock` action of the Clock component *reads* the value of `counter` which suggests that a mechanism for accessing the state variables in a component is required. Thus, we introduce the notion of a *getter* method which can be called on a component to produce a value that may be derived from its state variables. In the Counter component, the `getCounter` getter returns the current value of the counter. A getter is not allowed to modify the state of a component and may only be invoked in the immutable phase. These semantics preserve the strict separation of immutable phase and mutable phase. When analyzing composition, a getter is treated like

```
component Counter {
  var int counter (0)

  Tick: counter := counter + 1

  getCounter() int {
    return counter
  }
}
```

Figure 3.11: Definition of the Counter component of the Factored Clock System

```
component Clock {
  var Counter c
  var bool flag (false)
  push clock(int t)

  reaction request() flag := true

  Clock: flag -> flag := false activates clock(c.getCounter())
}
```

Figure 3.12: Definition of the Clock component of the Factored Clock System

a transition that reads the state variable of the corresponding instance. Figure 3.12 shows the Clock component rewritten to use a Counter sub-component and a getter.

The logic associated with the `flag` state variable represents a generic request-response protocol except for the call to `c.getCounter()`. To indirect the call to `c.getCounter()` we introduce the notion of a *pull port*. A pull port represents an external value dependency. A component can demand a value from the pull port in the immutable phase. Like push ports, pull ports have an active and passive side. The active side represents the caller and the passive side represents the callee. Getters are sufficient to realize the passive side of a pull port. Every active pull port must be bound to exactly one passive pull port via composition. Figure 3.13 shows a component that implements the request-response protocol using a pull port `getValue`. Figure 3.14 shows the Clock component written in terms of the Counter and RequestResponse components. An `export` directive allows reactions, getters, and ports in sub-components to be available in the interface of the encapsulating component.

```
component RequestResponse {
  var bool flag (false)
  pull getValue() int
  push response(int t)

  reaction request() flag := true

  Response: flag -> flag := false activates response(getValue())
}
```

Figure 3.13: Definition of the RequestResponse component of the Factored Clock System

```
component Clock {
  var Counter c
  var RequestResponse rr

  bind {
    c.getCounter -> rr.getValue
  }

  export rr.request as request
  export rr.response as clock
}
```

Figure 3.14: Definition of the Clock component of the Factored Clock System (fully-factored)

Pull ports are subject to a hazard of composition similar to the recursive activation hazard of push ports. A cycle in the graph of composed pull ports is equivalent to a recursively defined function. The recursion may be bounded but this is undecidable in the general case. Consequently, an implementation may reject recursively defined getters and pull ports.

3.4 Summary

In this chapter, we have presented the reactive component model for reactive programs. A reactive component consists of a set of state variables and transitions that are private to the component. The interface of a reactive component consists of push ports and reactions, which allow a component to trigger a transition in another component, and pull ports and getters, which allow a component to access the state of another component. The external or visible behavior of a reactive component can be traced through its interface, specifically its push ports. The internal details of a component can often be abstracted away to permit reasoning about the behavior of a composed system at various levels of detail. Composition is achieved through recursive encapsulation (sub-components) and explicit port binding and satisfies the requirements for principled composition set forth in Section 1.4. As demonstrated in Section 3.3.1, the definitions of sub-components can be substituted into the containing component resulting in an equivalent system (substitutional equivalence). Similarly, sub-components may be "factored out" by using pull ports and getters to safely access the state of the sub-components.

The private nature of state variables and transitions causes properties established from the text of a component to be preserved through composition. Composition links transitions to form an atomic transaction that allows the properties of one component to be related to another component. When the sub-components of a component are protected from further composition, the properties derived from their interactions are preserved as the parent component is composed. Property-preserving composition is essential for reasoning about systems in a hierarchical and/or modular fashion.

The result of composing reactive components is either well-defined due to the atomic nature of transactions or illegal due to the composition hazards of recursive transactions and non-deterministic state transitions. Analysis of these hazards at the state variable level is impossible due to the undecidable nature of their sub-problems. This suggests that implementations may restrict composition to prevent the conditions necessary for recursive transactions and non-deterministic state transitions. The main concession is allowing a component instance to proxy for its state variables. The problem of detecting recursive transactions, then, can be posed as the problem of detecting cycles in a directed graph. Similarly, the problem of detecting potentially non-deterministic state transitions is reduced to a set membership problem. Valid systems that fail the check for non-deterministic state transitions using component instance proxies can be refactored by decomposing the offending components.

Chapter 4

The $\mathrm{rc_{go}}$ Programming Language

In theory, there is no difference between theory and practice. But, in practice, there is. *Anonymous*

In this chapter, we present $\mathrm{rc_{go}}$, a novel extension to the Go programming language, designed for reactive components. We state the motivation for the $\mathrm{rc_{go}}$ language, our assumptions for tractability, and the features that guided our design. We then explain how reactive components are expressed in the language and we conclude with illustrative examples.

4.1 Challenges

An implementation of the reactive component model presented in Chapter 3 is necessary for at least two reasons. First and foremost, an implementation tests the practicality of the model. The act of implementing the model can help to evaluate whether the assumptions upon which the model is founded can be realized using existing techniques. Conversely, an implementation can suggest restrictions to the model that are necessary to produce an effective implementation. An example of this was seen in Section 3.3 where a component instance was used as a proxy for its state variables, for the purpose of determining which variables were involved in a transaction. Implementation forces one to supply and consider details that can either qualify or disqualify a model as a practical engineering tool. This is consistent with the emerging attitude in systems research that all new ideas and techniques must be accompanied by relevant tools and evaluations to show their feasibility [58].

Second, an implementation is necessary to demonstrate that the model can be applied successfully to real-world design and implementation problems. That is, given a platform for reactive components, we can design, construct, and evaluate systems based on the reactive component paradigm. Furthermore, we can evaluate critically the design and implementation processes that the model and platform encourage. By comparing implementations of similar systems in different models, we also can gain insight into the strengths and weaknesses of each model. These ideas will be explored further in Chapter 5.

4.2 Constraints

Our implementation of the reactive component model is shaped by a number of practical concerns. First, the implementation must enforce the semantics of the model to avoid subtle errors caused by reasoning about a system using one set of semantics and implementing it using another set of semantics. Second, the implementation must permit the use of linked data structures as they are fundamental to the efficient implementation of many algorithms. Third, the implementation must support reference and move semantics for efficient communication.

Strict enforcement of the reactive component model. To enforce the semantics of reactive components, the checks of interest are 1) the separation of the immutable and mutable phase (Section 3.1.5), 2) the binding of pull ports (Section 3.3.3), and 3) the detection of non-deterministic state transitions arising from composition (Section 3.3.2). Not performing these checks is unacceptable due to the subtlety associated with developing correct reactive programs: the semantics of reactive components would be enforced only through convention which is easily violated. Similarly, placing the burden on developers is unacceptable due to the amount of detail that must be considered. Thus, the implementation must enforce the semantics of reactive components through adequate checking.

Support for reference semantics and linked data structures. Formal models for reactive systems typically use *logic friendly* data-types, i.e., those that do not introduce aliasing, to make proofs easier. However, reference semantics and the linked data structures they make possible are a critical part of modern software engineering. As two of our objectives are practicality and utility, our language for reactive components must support reference semantics and linked data structures. The potential hazard created by reference semantics is that the state of one component becomes accessible in another component. This means that the state of a component may not be solely under the control of the transitions for that component. Consequently, the properties associated with that component could be violated by the other components manipulating that state. Furthermore, an implementation that chooses to execute seemingly independent transactions concurrently may cause data corruption since the concurrent transactions may manipulate the same state in an uncoordinated fashion. Thus, our implementation of reactive components must take special care to preserve isolation of state among reactive components when supporting references and linked data structures.

Efficient inter-component communication. Linked data structures also create an opportunity for efficient communication between components. There are three modes by which a component (sender) can share information with another component (receiver) as they interact through push ports and pull ports. First, the sender may use *value semantics* where it provides complete copies of the values to be communicated. This approach is reasonable for values like numbers and small records. Second, a sender may use *reference semantics* where it provides a pointer (reference) to the data to be communicated. This approach is reasonable when the sender offers up a large data structure. The receiver is responsible for copying any data that needs to be retained. When using such reference semantics, receivers may read and copy the data structure represented by the pointer but may not alter or persistently remember the original data structure in any way. Third, a sender may use *move semantics* where it provides a pointer to a data structure that the receiver may adopt as its own. In this case, the sender promises to "forget" the data structure as the receiver is the new owner of the data structure. Move semantics are appropriate when the size of the data to be communicated is large, there is a single receiver, and the sender does not need to retain the data. These conditions may arise, for example, in situations involving networking stacks and pipelines. Without move semantics, the practicality of the reactive component model is greatly diminished.

4.3 Approach

Our approach to implementing the reactive component model defined in Chapter 3 is to provide a programming language that facilitates the direct expression of reactive components. Language support for the model is beneficial because it closes the semantic gap between reasoning and implementation. The importance of language support can be seen in techniques like structured programming [29] and object-oriented programming [17]. While these techniques can be applied in virtually any setting, their lasting utility is derived from their implementation in a variety of programming languages. Providing language support for reactive components raises the level of abstraction and allows reasoning about a system consisting of them directly from its specification, instead of reasoning in one set of semantics while implementing in another which can be tedious and error-prone.

Language support allows developers to rely on the consistent application of the semantics of the model through strict enforcement. Designing language support specifically for reactive components creates an opportunity to introduce syntax and semantics that allow a compiler or interpreter to distinguish programs that conform to the semantics of reactive components from those that *potentially* may not. To illustrate, consider the problem of ensuring that the state of a component does not change during the immutable phase of a transition. We observe that an action/reaction is similar to a method. Based on this observation, checking the immutability of a component in the immutable phase should be similar to checking for `const` correctness in C++. In C++, the `const` correctness check is performed early in the compilation process when the program is represented as an abstract syntax tree (AST) with full semantic information, as opposed to late in the process where the program is represented by machine instructions with very limited semantic information. From this, we observe that such checking is supported by 1) adding features to the language to provide the necessary semantic information and 2) performing the checks early in the translation process. An existing language might not provide enough detail to enable the necessary checks or to achieve them efficiently. Along these same lines, an implementation may have to make conservative assumptions to enforce the semantics of reactive components. As was seen in Section 3.3, allowing a component instance to serve as a proxy for all of its state variables allows the check for non-deterministic state transitions to be implemented using known techniques.

4.4 Preliminaries

Our approach to programming language support for reactive components is to start with an existing language, Go [2], remove features that interfere with the semantics of reactive components, and then introduce syntax and semantics for reactive components. Go is an imperative programming language with a straightforward type system and expressions and statements resembling C. Go supports methods but places no emphasis on an inheritance hierarchy. In the same vein, Go does not have constructors, destructors, function overloading, or operator overloading. This combination makes Go an attractive foundation for a language for reactive components since it is tractable in implementation and approachable by a general audience. We defer to Go's syntax and semantics for our definitions of types, declarations, statements, and expressions. We will not discuss the syntax and semantics of Go except when they interact with those of reactive components. Our implementation of Go's types, declarations, statements, and expressions also is intentionally only as broad as is needed to demonstrate the contributions of this dissertation.

In $\mathrm{rc}_{\mathrm{go}}$, actions, reactions, getters, initializers, and bindings are expressed using Go's syntax for methods. Component types are constructed using the syntax of `struct`s. Push ports and pull ports are fields of a component. To enforce the immutable phase and facilitate checks for reference semantics, we introduce syntax that prevents the abuse of pointers. Activations in the model correspond to activate statements which activate push ports and contain the mutable phase of a state transitions. Move semantics are realized through a new *transferable heap* type and associated operations. The main features that we remove from Go are go routines, i.e., threads, and channels which are used for CSP-style communication.

Designing a programming language for reactive components using a different foundation language (or set of foundation languages) would require a different approach and may yield different results. For example, memory management in Go is based on garbage collection which is consistent with allowing reference semantics while isolating component state. In contrast, adopting a language based on manual memory management like C and C++ will require a reconsideration as to if/how such isolation may be achieved.

Static system assumption. To make implementation more tractable, we will assume that the systems to be implemented have a static topology, meaning that all reactive components are statically defined. Both finite state and infinite state (subject to system resource limits) reactive components are permitted, but both the number and configurations of reactive components in a system are fixed. This is roughly equivalent to systems that assume a fixed number of actors or threads. These assumptions are common in embedded and real-time systems due to the combination of limited resources and a need for predictability. These assumptions also are common in many other less constrained environments, as the number of threads is often fixed by the design, e.g., only a fixed number of concurrent activities is needed, or the number of threads is limited by the number of available physical cores [99]. Thus, even with the static system assumption, an implementation of the model is still applicable to many systems of interest. We leave the implementation of extensions that facilitate the dynamic creation and binding of reactive components for future work.

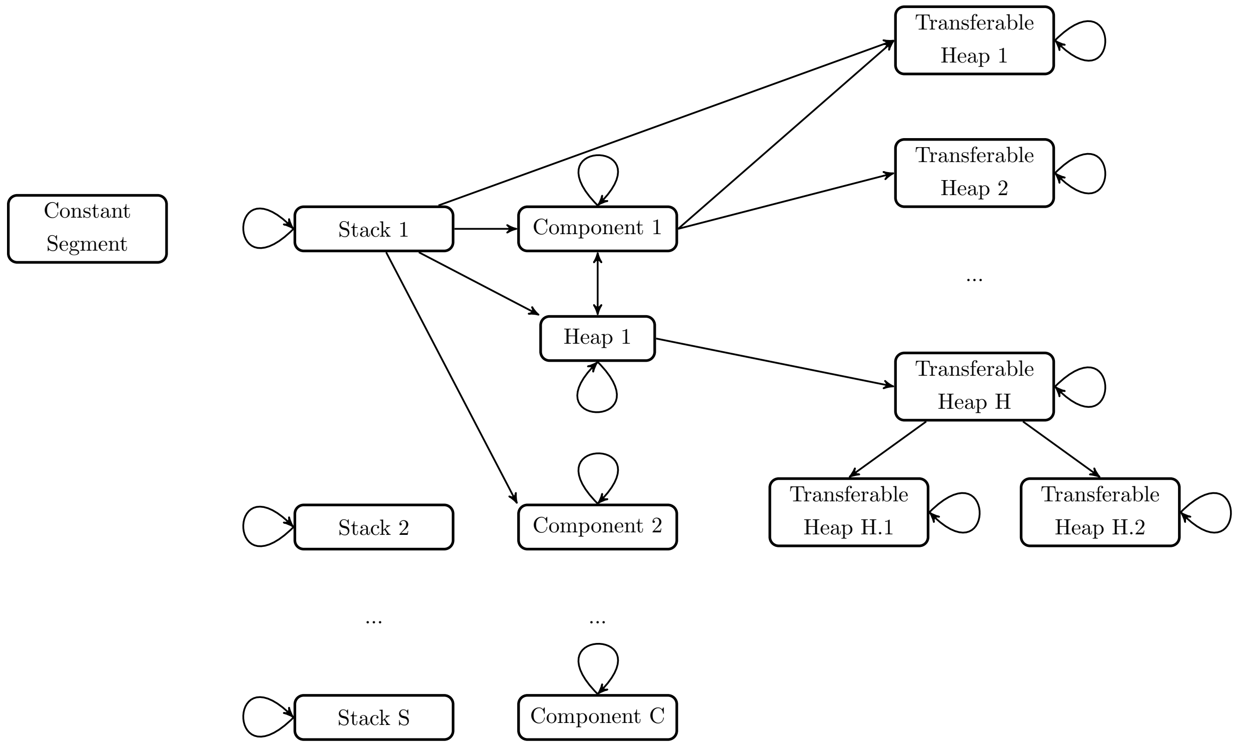

Figure 4.1: Memory model for the rc_{go} run-time system. An edge from one segment to another indicates that a pointer in one segment (source) may refer to a location in another segment (target). Any pointer may refer to a location in the constant segment (these edges are not shown).

Memory model. Figure 4.1 illustrates the memory model for rc_{go}. A pointer may refer to a location in the constant segment, a stack segment, a component segment, a heap segment, or a transferable heap segment. The edges in the figure indicate that a pointer in one segment may refer to a location in another segment. As implied by the name, the constant segment is used to store constants like string literals. A pointer in any segment may refer to the constant segment (these edges are not shown shown in the figure).

Function parameters and local variables are allocated on a call stack as they are in C or C++. Multiple stacks may be present if multiple transactions are executed concurrently. As indicated by the figure, a pointer in a stack segment may refer to a component segment, a heap segment, a transferable heap segment, or the constant segment. A pointer referring to a location on a call stack becomes invalid after the corresponding transaction is complete and the call stack is cleared. Consequently, it is not safe to allow static or dynamic component state to refer to a location in a stack as it may create a dangling pointer. Escape analysis [74] can be used to determine if an object should be dynamically allocated to avoid this problem. It is permissible to allow a pointer in a stack to refer to a location in the same stack.

As shown in Figure 4.1, the other three types of segments are component segments, heap segments, and transferable heap segments. A component segment contains the statically allocated state of a component while a heap segment contains the dynamically allocated state. A transferable heap segment is a dynamically allocated and self-contained group of objects that can be used to extend the state of a component and make such state transferable. The semantics of reactive components require the state of each component to remain disjoint. A pointer in a component segment may only refer to 1) a location in the constant segment, 2) a location in the same component segment, 3) a location in the corresponding heap segment, or 4) a transferable heap segment. Similarly, a pointer in a heap segment may only refer to 1) a location in the constant segment, 2) a location in the same heap segment, 3) a location in the corresponding component segment, or 4) a transferable heap segment. A pointer in a transferable heap segment may only refer to 1) a location in the constant segment, 2) a location in the same transferable heap segment, or 3) another transferable heap segment. Part of our approach to maintaining disjoint state is preventing the formation of arbitrary pointers that would allow one component to access the state of another component. Consequently, pointer arithmetic and casts from numeric values are not allowed. Manually deallocating memory may result in dangling pointers which, as memory is recycled, may point to the state of another component. Consequently, some form of automatic memory management is necessary with the two primary candidates being reference counting and garbage collection. Our implementation uses garbage collection and is described in Section 5.4. The component segments, heap segments, and transferable heap segments contain all of the mutable state in the system. Mutable state outside of a component, e.g., a global variable, is prohibited as it may introduce a data race.

4.5 Syntax and Semantics

This section describes the syntax and semantics of rc$_{\text{go}}$. Many concepts, e.g., components, ports, actions, and reactions, have a natural mapping into the Go language as types and method-like constructs. A major divergence from Go is the introduction of attributes that change the mutability of variables, as this was necessary to maintain the isolation between components while supporting reference semantics. The separation between the immutable phase and mutable phase is accomplished via an `activate` statement that contains a continuation to be executed in the mutable phase. To support move semantics, we introduce a new type called a *transferable heap* which allows one component to safely transfer a collection of objects to another component. In the following presentation, we use the following convention:

- keywords appear in lowercase, e.g., `type`
- identifiers and literals appear in uppercase and underscores, e.g., `ID`
- non-terminals appear in camel-case, e.g., `FieldList`

Components. A reactive component resembles a `struct` in Go since it is a group of named state variables. A component, then, is defined with the following syntax:

```
type ID component { FieldList };
```

For example,

```
type Clock component {
  flag bool;
  counter uint;
};
```

introduces a type named `Clock` that is a reactive component with two fields (state variables). The type of a field may be another component type to support recursive encapsulation.

Receivers. A method in Go has one of the following two forms:

```
func (ID TYPE_ID) METHOD_ID Signature Body
func (ID *TYPE_ID) METHOD_ID Signature Body
```

The first form operates on a copy of a value of type `TYPE_ID`. The second form operates on a pointer to a value of type `TYPE_ID`. The parameter `ID` names the receiver of the method and performs the same function as the `this` keyword in C++ and Java. The syntax `(ID TYPE_ID)` is called a *receiver* and the syntax `(ID *TYPE_ID)` is called a *pointer receiver*. Actions, reactions, getters, and initializers use a pointer receiver.

Intrinsic and indirection mutability. Variables and parameters are declared with both an *intrinsic mutability* and a *indirection mutability*. Intrinsic mutability limits the operations that can be performed on the lvalue of the variable or parameter while indirection mutability limits the operations that can be performed on lvalues derived from the rvalue of the variable or parameter. By default, variables and parameters have mutable intrinsic and indirection mutability. The following code is legal and sets `z` to 6.

```
var x uint = 3;
var y *uint = &x;
var z = x + *y;
```

Declaring a variable or parameter to have immutable intrinsic mutability (`const`) prevents the variable or parameter from being changed after it is initialized. The following code is illegal:

```
var x const uint = 3;
x = 4; // Illegal
```

Immutable intrinsic mutability is enforced when taking the address of a variable or parameter:

```
var x const uint = 3;
var y *uint = &x;        // Illegal
var z $const *uint = &x; // Legal
var a uint = *z;         // Legal
*z = 5;                  // Illegal
```

The second line is illegal because `x` could change through a statement like `*y = 5;`. The third line causes `z` to have immutable indirection mutability (`$const`). The expression `*z` can serve as an rvalue (line 4) but it cannot serve as an lvalue (line 5).

Indirection mutability is "sticky." For example:

```
var x $const **uint = ...;
var y $const *uint = *x; // Legal
var z *uint = *x;        // Illegal
```

The second line honors the guarantee that all of the memory accessible through `x` is immutable. Indirection mutability is checked in assignments and calls. Indirection mutability affects types from which lvalues can be derived, namely, pointers and slices[3]. Immutable indirection mutability is one of the techniques that is used to enforce the immutable phase of state transitions.

The second kind of mutability is called *foreign* mutability. Foreign mutability is like immutable mutability with the added condition that an address with foreign mutability cannot be stored in a component segment, heap segment, or transferable heap segment. The primary application of foreign mutability is to support reference semantics for communication while enforcing the isolation of heaps.

```
var x $foreign *uint = ...;
var y **uint = new (*uint);
*x = 3;                            // Illegal
*y = x;                            // Illegal
var z $foreign *uint = x;          // Legal
```

[3]A *slice* represents a portion of an array and consists of a pointer, a size, and a capacity.

The third line of the code fragment above is illegal because the lvalue given by `*x` is immutable. The fourth line is illegal because it casts away the `$foreign` attribute of `x`. (Notice that declaring `y` with `$foreign` would cause the lvalue to be immutable.) The fifth line is legal because the lvalue is mutable and the `$foreign` attribute is preserved. The consequence of these semantics is that variables that contain pointers that are declared `$foreign` may only be stored in stack segments.

The following checks are applied to assignment statements:

1. The lvalue and rvalue must be type compatible.
2. The lvalue must have mutable intrinsic mutability.
3. If the type involved contains a pointer, check for compatible indirection mutability (see Table 4.1). Essentially, the indirection mutability of the lvalue must be at least as "weak" as the indirection mutability of the rvalue. This enforces the "stickiness" of indirection mutability.

	Mutable	Immutable	Foreign
Mutable	Yes	No	No
Immutable	Yes	Yes	No
Foreign	Yes	Yes	Yes

Table 4.1: Indirection mutability compatibility for assignment. The rows represent the indirection mutability of the lvalue and the columns represent the indirection mutability of the rvalue.

A parameter is *foreign safe* if 1) the type of the parameter does not contain pointers or slices or 2) the parameter is declared with foreign indirection immutability. A parameter list is foreign safe if all parameters in the list are foreign safe. A signature (a parameter list and a return parameter list) is foreign safe if the parameter list and return parameter list are both foreign safe. Signatures used in inter-component communication, such as push ports and reactions, must be foreign safe to permit reference semantics while enforcing the isolation of state between components.

Initializers. An initializer is used to initialize the fields of a reactive component. This allows one to initialize components before the scheduler starts. This is necessary for establishing invariants as is commonly done in formal models, e.g., the `initially` section of UNITY [20]. An initializer has the form:

```
init (ID *TYPE_ID) INITIALIZER_ID Signature Body
```

An initializer is similar to a method but has additional semantics:

- An initializer must have a pointer receiver to a component type.
- The signature must be foreign safe.
- An initializer may only be invoked by another initializer.
- An initializer sets the heap segment on entry and resets the heap segment on exit so that all allocated memory is attributed to the receiver.

Instances. An instance is a top-level component. An instance is declared with the following syntax:

```
instance ID TYPE_ID INITIALIZER_ID (ExpressionList);
```

For example, `instance c Clock Init ();` declares as instance named `c` of type `Clock` and will call the initializer `Init` with an empty list of arguments. The instance identifier must be unique, the type identifier must refer to a component, and the initializer must be declared for the component type.

Actions. An action has the form:

```
action (ID $const *TYPE_ID) ACTION_ID (BooleanExpression) Body
```

The immutable indirection mutability of the receiver enforces the immutable phase of transitions. Actions may only be defined for component types. The Boolean expression is the precondition of the action. The receiver variable is in scope for the evaluation of the precondition. The body contains the state transitions associated with the action. Actions set the heap segment on entry and reset the heap segment on exit so that all allocated memory is attributed to the receiver.

Reactions. A reaction has the form:

```
reaction (ID $const *TYPE_ID) REACTION_ID (ParameterList) Body
```

As with actions, the immutable indirection mutability of the receiver enforces the immutable phase of transitions. Reactions may only be defined for component types. The name of a reaction is used when binding to push ports. The parameter list declares the parameters that are passed to the reaction. The parameter list must be foreign safe. This prevents the reaction from storing memory addresses from the component that activated the reaction. The body contains the state transitions associated with the reaction. Reactions set the heap segment on entry and reset the heap segment on exit so that all allocated memory is attributed to the receiver.

Push ports. A push port is declared as a field (of a component) with push port type. A push port type has the form:

```
push (ParameterList)
```

The parameter list declares the parameters that are passed to any bound reaction. The parameter list must be foreign safe. The following example declares a push port named `response` in the `Clock` component:

```
type Clock component {
  ...
  response push (t uint);
};
```

Getters. A getter, which provides a safe way of obtaining information from a component, has the form:

```
getter (ID $const *TYPE_ID) GETTER_ID Signature Body
```

A getter is similar to a method but has additional semantics:

- A getter must have a pointer receiver to a component type declared with immutable indirection mutability.
- The signature must be foreign safe.
- A getter may only be invoked by an initializer, another getter, or an action or reaction in the immutable phase.
- A getter sets the heap on entry and resets the heap on exit so that all allocated memory is attributed to the receiver.

Pull ports. A pull port is declared as a field (of a component) with pull port type. A pull port type has the form:

```
pull (ParameterList) ReturnParameterList
```

The parameter list declares the parameters that are passed to the bound getter and the return parameter list declares the return values of the getter. The parameter list and return parameter list must be foreign safe. Pull ports are called like getters and place the same restriction on the caller, that is, a pull port may only be invoked by a getter or an action or reaction in the immutable phase. The following example declares a pull port named `isOutputBufferFull` in the `Producer` component:

```
type Producer component {
  ...
  isOutputBufferFull pull () bool;
};
```

In this example, the intent of the pull port is to allow a `Producer` to interrogate the status of a downstream buffer to implement flow control.

Binders. Binders allow reactions to be associated with push ports and getters to be associated with pull ports. Binding and recursive encapsulation are the two mechanisms for composing reactive components. A binder has the form:

```
bind (ID *TYPE_ID) BIND_ID {
  BindStatement
  ...
}
```

For example:

```
bind (this *System) TheBinder {
  this.producer.Out -> this.consumer.In;
  this.producer.isOutputBufferFull <- this.consumer.isInputBufferFull;
}
```

The first statement of the example binds the `Out` push port of the `System`'s `producer` to the `In` reaction of the `System`'s `consumer`[4]. The second statement of the example binds the `isOutputBufferFull` pull port of the `System`'s `producer` to the `isInputBufferFull` getter of the `System`'s `consumer`. The left side of a bind statement always refers to a port while the right side refers to a getter or reaction. The direction of the arrow indicates the logical flow of information. Thus, information flows from a push port to a reaction (`->`) and information flows from a getter to a pull port (`<-`). A pull port must be bound to exactly one getter. A reaction may be bound to at most one push port. Binders are associated with a component type and evaluated for each instance of that component type.

[4]The select operator(`.`) automatically dereferences pointers.

Activations. Activations are the mechanism by which transitions extend to other components via push port/reaction bindings. Activations also serve as the boundary between the immutable and mutable phases of a transition. An activate statement has the form:

```
activate PORT_ID (Arguments) ... {
  Statements
};
```

Activate statements can only occur in the body of an action or a reaction. Assume that the receiver of the action or reaction is named `this`[5]. The expression `this.PORT_ID` must refer to a push port and the arguments passed to the push port must agree with its signature. The list of push ports in an activate statement is optional. When an activate statement is executed, the named push ports are activated meaning that the reactions bound to those push ports are executed with the given arguments. These reactions, in turn, may execute other activate statements. Once all of the actions and reactions in the transaction have activated their last push ports, they proceed to execute the bodies of the activate statements. Recall that the receiver `this` has immutable indirection mutability. Thus, all computation up to and including the last port activation constitutes the immutable phase of the transaction since the state of the components is not allowed to change. Within the scope of the body, the receiver `this` changes to mutable indirection mutability which then allows the state of a component to be changed. Thus, the bodies of activate statements form the mutable phase of the transaction. Parameters and variables declared with foreign indirection mutability are hidden within the body of an activate statement. This prevents one component from accessing the state of another component during the mutable phase. Actions and reactions return, i.e., their flow of control is halted, after the execution of the body of a activate statement. Activate statements guarded by `if` statements facilitate *conditional activation*. An activate statement may not appear in another activate statement. Our implementation of activate statements is described in Section 5.3.

Arrays. A homogeneous group of sub-components may be declared using array syntax. For example,

[5]`this` is not a keyword in Go.

```
type System component {
  clock [5]Clock;
  ...
};
```

declares 5 `Clock` sub-components. To request the time from each `Clock`, the `System` declares an array of 5 push ports and a *dimensioned* action:

```
type System component {
  clock [5]Clock;
  push [5]request ();
  ...
};

[5] action (this $const *System) (...) {
  ...
  activate clock[IOTA] {
    ...
  };
}
```

A dimensioned action is parameterized with an integral constant in the range $[0, \mathit{dimension})$. This constant is accessed through the `IOTA` symbol. A push port in an array is activated by supplying an index. The index expression must be constant to facilitate the check for sound composition. Reactions may be dimensioned as well:

```
[5] reaction (this $const *System) clock (t int) { ... }
```

A `for`-loop over an integral range may be used to generate bindings without explicitly listing each binding. For example:

```
bind (this *System) {
  for i ... 5 {
    this.request[i] -> this.clock[i].request;
  };
}
```

Transferable heaps. A key requirement for implementing reactive components is that the state of each component remain disjoint. Foreign indirection mutability allows components to safely communicate with pointers because it ensures that those pointers are forgotten after the transaction. For efficient communication, we also desire the ability to transfer a data structure (a heap) from one component (the sender) to another component (the receiver). The sender offers the heap to its receivers and one of the receivers may claim the heap. If the heap is accepted, the sender must forget all references to the heap.

The transferable heap type is so named because it resembles a heap used for dynamic memory allocation. A heap has a distinguished root that contains the data structure that will be transferred. A heap is entirely self-contained, that is, any pointer found in the heap may only point to an address in the heap or another transferable heap segment. This ensures that the receiver may not access state in the sender after a transfer. Heaps may form hierarchies.

A heap is created with the `new` operator. For example:

```
var x *heap int = new (heap int);
```

creates new heap with an integer root.

A `change` statement allows one to access the root of the heap. For example:

```
change (x, y) {
  *y = 3;
};
```

In the example, `x` is a pointer to a heap and `y` is a variable that points to the root of the heap. The root variable is valid for the scope introduced by the `change` statement. The root variable will be set to `nil` if the heap is no longer valid. Within the scope of the `change` statement, all other variables and parameters that may contain pointers are re-entered with foreign indirection immutability. This enforces the isolation of heaps by preventing the heap from storing a pointer that refers to a location in another heap.

Logically, the run-time system maintains a stack of heaps. The top of the stack is called the *active heap* and is used to service all memory allocation requests. On entering an action or reaction, the stack of heaps contains a single heap: the heap associated with the receiver component. A `change` statement pushes a new heap on the stack.

A `move` expression allows a receiver to take ownership of a heap being offered by a sender. For example:

```
var z *heap int = move (x);
```

If `x` refers to a heap that has already been moved, i.e., claimed by this component or another component, then the result of the `move` is a `nil` pointer. A `change` statement can be used to access the data in the heap after a successful `move`.

A `merge` expression allows one to merge a heap into the active heap. For example:

```
var x *uint = merge (z);
```

A `merge` expression performs an implicit `move`, that is, the heap given to `merge` need not be owned by the current component. Similarly, a `merge` will return `nil` if the heap has already been claimed.

Operations on heaps, specifically, `change`, `move`, and `merge` are atomic within a transaction. The reactive component model requires a clear distinction between the mutable phase and immutable phase and some causality in the immutable phase. However, an implementation is free to execute immutable phases concurrently and/or mutable phases concurrently. Consequently, different components may be performing heap operations on the same heap at the same time. Thus, `change`, `move`, and `merge` are atomic with respect to each other. These operations return `nil` if they fail. For example, if two components attempt to `move` the same heap, one will succeed and the other will fail.

4.6 Examples

To demonstrate the features of rc$_{\text{go}}$, we present two examples. The first example simulates three users using three processes which communicate using a shared variable. This example shows all of the major syntactic elements such as component types, initializers, actions, reactions, getters, push ports, pull ports, binders, activate statements, arrays, and `$const`. The second example consists of a system and a channel. The system sends an object to the channel and then receives the object from the channel. This example demonstrates the transferable heap type, `$foreign`, `new`, `change`, `move`, and `merge`.

4.6.1 Shared Variable System

In this example, we rewrite the shared variable I/O automaton on pages 240 and 242 of [64] as a reactive component. The example consists of four kinds of components. First, there is a `Variable` component which represents a shared variable. `Process` components access the shared variable and report the status of the shared variable to `User` components. A top-level `System` component instantiates the shared variable, three processes, and three users to create a complete system. Figure 4.2 shows the `Variable` component.

```
type Variable component {
    value int;
};

init (this *Variable) Init () {
    this.value = -1;
}

[3] reaction (this $const * Variable) Set (v int) {
    activate {
        this.value = v;
    }
}

getter (this $const * Variable) Get () int {
    return this.value;
}
```

Figure 4.2: Code listing for `Variable` component of the Shared Variable System

A `Variable` component has a single integer state variable named `value`. This variable is initialized to the sentinel value -1 in the initializer `Init`. Notice that the initializer has a pointer receiver `(this *Variable)`. The receiver `this` has mutable indirection mutability so that the state variable can be assigned. The value of the `Variable` is set in the dimensioned reaction `Set`. The reaction has a dimension of three as three `Process` actions will be bound to this reaction. The new value for the variable is communicated via the parameter `v`. Notice that the receiver for the reaction `(this $const * Variable)` has immutable indirection mutability. In the body of the activate statement, the receiver is implicitly converted to have mutable indirection mutability. The final element of the `Variable` component is the getter `Get` that returns the value of the variable. Since getters are not allowed to change the state of the component, they must be declared with immutable indirection mutability `(this $const * Variable)`.

Figures 4.3 and 4.4 shows the `Process` component. The listing begins by defining the states for a process, that is, a process is either idle, ready to access the shared variable, ready to decide, or done. A `Process` component consists of three state variables: `status` contains the state of the process, `input` contains a value received from the `User`, and `output` contains the value that will be sent to the `User`. The `Process` component contains a pull port `get_x` that will be bound to the shared variable's `Get` getter so that the `Process` may access the shared variable. Similarly, the `Process` component contains a `set_x` push port that will be bound to the shared variable's `Set` reaction to set the value of the variable. To communicate with the `User`, the `Process` component contains a push port `Decide` that communicates the value of the shared variable.

Like the `Variable` component, the `Process` component contains an initializer `Init` that sets the initial value of the state variables. From the listing, `Process` components are initialized to the idle state and the `input` and `output` variables are initialized to the sentinel value of -1. The `initr` reaction allows a `User` to asynchronously initialize the `Process` component. From the listing, this reaction sets the `input` variable to the value supplied in the argument `v` and causes the `Process` to prepare to access the shared variable if the `Process` was previously idle.

```
type ProcessStatus int;
const PROCESS_IDLE = 0;
const PROCESS_ACCESS = 1;
const PROCESS_DECIDE = 2;
const PROCESS_DONE = 3;

type Process component {
    status ProcessStatus;
    input int;
    output int;
    get_x pull () int;
    set_x push (v int);
    Decide push (v int);
};

init (this *Process) Init () {
    this.status = PROCESS_IDLE;
    this.input = -1;
    this.output = -1;
}

reaction (this $const * Process) initr (v int) {
    activate {
        this.input = v;
        if this.status == PROCESS_IDLE {
            this.status = PROCESS_ACCESS;
        }
    }
}
```

Figure 4.3: Code listing for `Process` component of the Shared Variable System (part 1)

```
action (this $const * Process) _access (this.status == PROCESS_ACCESS) {
    x := this.get_x ();
    if x == -1 {
        activate set_x (this.input) {
            println (‘x set to ‘, this.input);
            this.output = this.input;
            this.status = PROCESS_DECIDE;
        }
    }
    else {
        activate {
            this.output = x;
            this.status = PROCESS_DECIDE;
        }
    }
}

action (this $const * Process) _decide (this.status == PROCESS_DECIDE) {
    activate Decide (this.output) {
        this.status = PROCESS_DONE;
    }
}
```

Figure 4.4: Code listing for `Process` component of the Shared Variable System (part 2)

The most interesting part of the Process component is the _access action. Like reactions, actions honor the immutable phase by requiring a receiver with immutable indirection mutability (this $const * Process). The precondition for _access tests that the Process is ready to access the shared variable. When executed, the _access action samples the state of the shared variable by calling the get_x pull port and assigns this to the local variable x. Note that pull ports may only be called in the immutable phase. If the shared variable has not been set as indicated by the sentinel value -1, the process sets it to the value requested by the User (this.input) by activating the set_x push port. Once the mutable phase begins, the Process outputs a message, copies the input variable to the output variable, and then changes state to prepare to decide. If the shared variable has been set, i.e., another process executed its _access action first, the process sets its output variable to the value of the shared variable as stored in x and prepares to decide.

The final element in the Process component is the _decide action which activates the Decide push port with the value of the shared variable stored in the output variable and transitions to the done state.

The interface of the Process component gives hints as to its contextual dependencies and intended use. The Process component expects an initialization message (initr reaction) from the User and will report back to the User (Decide push port). The Process component requires the ability to interrogate the value of the shared variable (get_x pull port) and the ability to the set the shared variable (set_x push port).

The state transitions of a Process component can be determined by tracing the status variable. A Process component starts in the idle state according to the Init initializer. The Process will stay idle until its initr reaction is activated at which point it will enter the access state. Given the fairness guarantees of the scheduler, the _access action will eventually be executed as the Process is in the access state. The execution of _access is governed by an if statement but both branches set the state of the Process component to decide. Given the same fairness guarantees, the _decide action will eventually be executed as the Process is in the decide state. The _decide action unconditionally sets the state of the component to done. There is no way for a Process component to leave the done state as all of its actions are conditioned on the Process being in either the access or decide state. Similarly, the initr reaction only moves the Process from the idle to the access state. Thus, the state of the component logically flows from idle, to access, to decide, to done assuming that the `User` does indeed activate the `initr` reaction.

The preceding two paragraphs illustrate how the reactive component model and the proposed programming language achieve principled composition. The behavior of a component can be determined by only examining the text of the component. This is possible due to the strong guarantees that 1) component state cannot be manipulated outside of an action or reaction, 2) actions and reactions are logically atomic, and 3) the scheduler will eventually execute all enabled actions. For compositional reasoning, the behavior of a component can be abstracted and stated in terms of assumptions and guarantees [55] about its interface elements, namely, push ports, reaction, pull ports, and getters. For example, a `Process` component will decide the value of the shared variable when initialized via the `initr` reaction.

Figure 4.5 shows the `User` component. The `User` component has three states indicating the user is waiting to make a request, waiting for a response, or done. The first state variable `v` contains the value sent in the request to a `User`. The `status` state variable contains the state of the `User`. The `decision` state variable contains the value of the response. The `error` state variable is a flag indicating that an error has occurred. `User` components contain a push port `initp` that sends the requested value to the corresponding `Process`. The `Init` initializer sets the requested value to the supplied parameter, the state of the `User` to request, the `decision` variable to the sentinel value of -1, and the `error` flag to false.

The `_init` action shown in Figure 4.6 moves the `User` component from the request state to the waiting state while initializing the corresponding `Process` component via the `initp` push port. The `_dummy` action and the treatment of the `error` flag by the `_init` action are for consistency with [64]. The `Decide` reaction prints out the identity of the `User` and the value received by the `User`. If the `User` is not in an error condition and waiting for a response, then the value of the decision is recorded and the `User` changes to the done state. Otherwise, the `error` flag is set meaning that the corresponding `Process` activated the `Decide` reaction before the `_init` action.

```
type UserStatus int;
const USER_REQUEST = 0;
const USER_WAIT = 1;
const USER_DONE = 2;

type User component {
    v int;
    status UserStatus;
    decision int;
    error bool;
    initp push (v int);
};

init (this *User) Init (v int) {
    this.v = v;
    this.status = USER_REQUEST;
    this.decision = -1;
    this.error = false;
}
```

Figure 4.5: Code listing for `User` component of the Shared Variable System (part 1)

```
action (this $const * User) _init (this.status == USER_REQUEST || this.error) {
    activate initp (this.v) {
        if !this.error {
            this.status = USER_WAIT;
        }
    }
}

action (this $const * User) _dummy (this.error) { }

reaction (this $const * User) Decide (v int) {
    println (this, ‘ decided value is ‘, v);
    activate {
        if !this.error {
            if this.status == USER_WAIT {
                this.decision = v;
                this.status = USER_DONE;
            } else {
                this.error = true;
            }
        }
    }
}
```

Figure 4.6: Code listing for `User` component of the Shared Variable System (part 2)

```
type System component {
    x Variable;
    process [3]Process;
    user [3]User;
};

init (this *System) Init () {
    this.x.Init ();
    for i ... 3 {
        this.process[i].Init ();
        this.user[i].Init (i + 100);
    }
}

bind (this *System) _bind {
    for i ... 3 {
        this.process[i].get_x <- this.x.Get;
        this.process[i].set_x -> this.x.Set ... i;
        this.user[i].initp -> this.process[i].initr;
        this.process[i].Decide -> this.user[i].Decide;
    }
}

instance s System Init ();
```

Figure 4.7: Code listing for `System` component of the Shared Variable System

Figure 4.7 shows the `System` component. The system component contains a `Variable` sub-component, three `Process` sub-components, and three `User` sub-components. The `Init` initializer initializes all of the sub-components. The `User` processes are initialized with the values 100, 101, and 102. Thus, `User` 0 will attempt to set the `Variable` to 100, `User` 1 will attempt to set the `Variable` to 101, and `User` 2 will attempt to set the `Variable` to 102. The `_bind` binder "wires" the system. The first line of the `for` loop binds the `get_x` pull port of each `Process` to the `Get` getter of the `Variable`. The second line of the `for` loop binds the `set_x` push port of each `Process` to the `Set` reaction of the `Variable`. Notice that the `Set` reaction is indexed to avoid binding the same input multiple times. The third line of the `for` loop binds the `initp` push port of each `User` to the `initr` reaction of each `Process`. The fourth line of the `for` loop binds the `Decide` push port of each `Process` to the

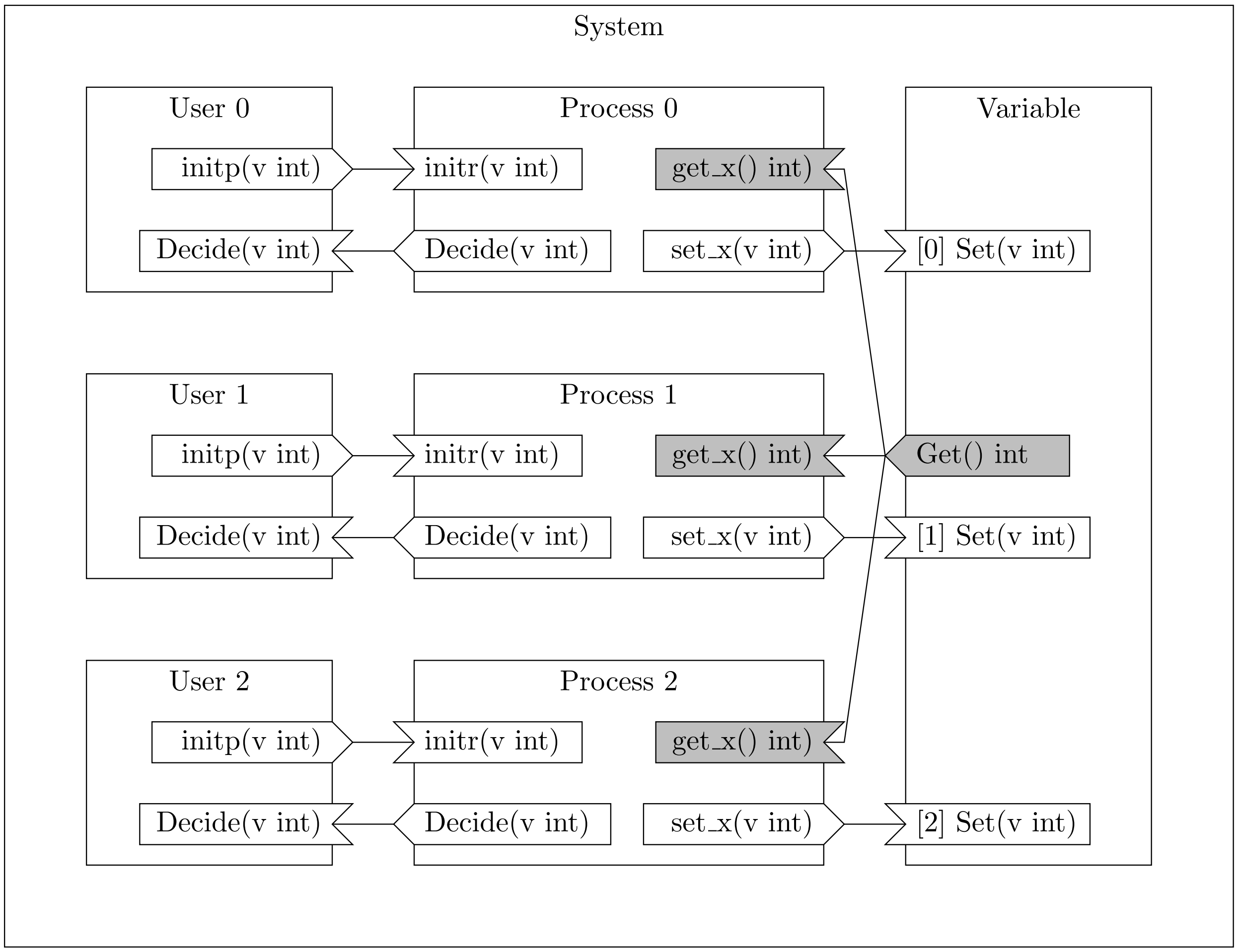

Figure 4.8: Diagram of a `System` component of the Shared Variable System

`Decide` reaction of each `User`. The final line of the listing creates an instance of the `System` component named `s` and initializes it with the `Init` initializer. Figure 4.8 shows a graphical representation of a `System` component.

Figure 4.9 shows sample output for an execution of a `System` component. The first line of output is generated by the `_access` action of the `Process` component. In the sample, `Process 2` is the first process to set the `Variable`. As expected, only one `Process` sets the variable. The final three lines of output show the identity of each `User` and the value returned to it. As expected, the value returned to each `User` is consistent with how the `Variable` was set.

```
x set to 102
0x1fd7258 decided value is 102
0x1fd71f8 decided value is 102
0x1fd7228 decided value is 102
```

Figure 4.9: Sample output for a `System` instance of the Shared Variable System

```
type Channel component {
  queue Queue;
  receive push (message $foreign *heap uint);
};

reaction (this $const * Channel) send (message $foreign *heap uint) {
  var x *heap uint = move (message);
  activate {
    this.queue.Push (x);
  };
}

action (this $const * Channel) _receive (!this.queue.Empty ()) {
  activate receive (this.queue.Front ()) {
    this.queue.Pop ();
  };
}
```

Figure 4.10: Code listing for `Channel` component of the Heap Channel System

4.6.2 Heap Channel System

In this example, we demonstrate how the `heap` data type may be used to transfer objects between components for efficient communication. The example consists of two components: a top-level `System` component and a `Channel` component. The `Channel` component is a reliable FIFO channel based on the Channel Automaton of [64]. The `System` component transfers 100 messages to the `Channel` component which then transfers the same 100 messages back to the `System`. The messages consists of a `heap` with a `uint` root object.

Figure 4.10 shows the `Channel` component. A `Channel` component consists of a queue of messages and a push port named `receive` which offers up a pointer to a heap with a `uint` root. Since the parameter contains (is) a pointer, it must be declared with `$foreign` indirection mutability. The elements of the `Channel` are named from the perspective of the process that uses the `Channel`.

The `send` reaction moves the heap which makes the `Channel` the new owner of the heap. Any heap operations on `message` after the move statement will return `nil`. The move occurs outside of the activation because the `message` parameter is not available in the body of the activation. Parameters and variables with types that contain pointers and have foreign indirection mutability are not available in the body of an activate statement because they may represent the state of another component which must be assumed to be invalid in the mutable phase of a transaction. After a move, the state contained in a heap is no longer available to the component that offered it up as an argument to a reaction. Thus, the result of the move does not have foreign indirection mutability and is available in the body of the activate statement.

Figure 4.11 shows the `System` component that exercises the `Channel`. A `System` component consists of a `Channel` sub-component, a counter, and a push port for sending messages. The `_send` action checks if the desired number of messages (100) has been sent. If not, the `System` creates a new heap with a `uint` root. It then makes the new heap the active heap via the `change` statement which also makes the root of the heap available in the variable `y`. The heap root is initialized with the number of messages sent so far. The heap is then sent to the `Channel`, a message is printed, and the number of sent messages is incremented.

The `receive` reaction receives a message back from the `Channel`. The `System` merges the message which moves the root object of a heap to the active heap and returns a pointer to the root object of the heap. Recall that all components have a default heap which is the active heap upon entering an action, reaction, initializer, or getter. The final statement in the reaction prints a message.

The system consists of two transactions: one that creates and transfers the heap from the `System` to the `Channel` and another that transfers the heap from the `Channel` to the `System`. The output of the program, then, consists of 100 lines indicating that a message was sent and 100 lines indicating that a message was received. Each line of output contains the message number. Based on the fairness of the scheduler we expect 1) the 100 send lines to be in

```
type System component {
  channel Channel;
  send_count uint;
  send push (message $foreign *heap uint);
};

init (this *System) Initially () { }

action (this $const * System) _send (this.send_count != 100) {
  var x *heap uint = new (heap uint);
  change (x, y) {
    *y = this.send_count;
  };
  activate send (x) {
    println (‘Sent ‘, this.send_count);
    this.send_count++;
  };
}

reaction (this $const * System) receive (message $foreign *heap uint) {
  var x *uint = merge (message);
  println (‘Received ‘, *x);
}

bind (this *System) Bind {
  this.send -> this.channel.send;
  this.channel.receive -> this.receive;
}

instance s System Initially ();
```

Figure 4.11: Code listing for `System` component of the Heap Channel System

order, 2) the 100 receive lines to be in order, and 3) the send line for message n appears before the receive line for message n.

4.7 Related Work

rc$_{\text{go}}$ is designed to be free from data races and draws upon existing work in this area. Common techniques include 1) augmenting type declarations, signatures, etc., to indicate sharing/locking requirements and effects; 2) associating objects with an owning thread or memory region; and 3) transferring objects from one owner to another. The `foreign` attribute of rc$_{\text{go}}$ is similar to the `lent` attribute of Guava [12] and the `limited` attribute of Promises [19]. All three techniques allow a reference to be temporarily shared but not stored so as to create a shared resource. The transferable heap data type is similar to Islands [53] and Values in Guava [12]. All three types represent object graphs with no external references which allow them to be transferred from one owner to another owner. In rc$_{\text{go}}$, objects are owned by component instances while Islands and Values are owned by threads.

Preventing data races in multi-threaded programs is accomplished by protecting critical sections with locks. Flanagan and Abadi developed a type system for statically checking for data races [37, 6]. This formalism was used in the design of both Guava [12] and Cyclone [47]. Locks are explicit in Cyclone while they are implicitly associated with the monitors of Guava. The reactive components of rc$_{\text{go}}$ are similar to the monitors of Guava in that access to shared state is implicitly synchronized.

The approach taken by the languages cited thus far is to demonstrate freedom from data races through the type system exclusively. The type system of rc$_{\text{go}}$ is weaker than these languages in that it is only possible to prove that objects are not shared between components via `foreign` indirection mutability. The check for sound composition alluded to in Section 3.3.2 and described in Section 5.2 ensures that individual transactions are free from data races. As described in Section 3.1.5 and Chapter 6, the scheduler has the responsibility of scheduling transactions in a way that avoids data races between transactions.

4.8 Summary

This chapter has presented the rc_{go}programming language for reactive components. An implementation of reactive components tests whether the assumptions upon which the model rests are practical. Our approach is to design new programming language syntax and semantics to capture and enforce the semantics of reactive components. As a matter of practicality, we impose support for reference semantics which allow developers to use linked data structures and support for transferring data structures from one component to another. The corresponding features are a declarable indirection mutability that allows components to treat pointers as foreign and a transferable heap data type that represents a self-contained linked data structure. Components are expressed as a collection of fields (similar to a `struct`). Ports (push and pull) are expressed as fields of a component. Actions and reactions are expressed as method-like elements. Composition is accomplished via binders and instances. Our implementation of an interpreter for $\text{rc}_{\text{go}}$is described in Section 5.1.

The rc_{go} programming language presented in this chapter allows developers to take a principled approach to developing general purpose reactive programs. First, a developer need not identify shared state and add appropriate locking because state is not shared between components and each action/reaction is atomic. Second, a developer need not worry about the consequences of composition. In paradigms where composition is accomplished through synchronous function call, developers must determine the conditions in which it is safe to transfer control to a function. In the reactive component paradigm, the semantics of activate statements and ports provide strong guarantees when composing and checking for illegal composition, which is the responsibility of the run-time system (Section 5.2). Third, developers are no longer burdened with mapping an inherently non-deterministic sequence of events onto one or more sequential threads of control. Combined, the features of the model and language allow developers to reason about the behavior of individual components by only examining their text. This, in turn, allows the behavior of a component to be abstracted which facilitates reasoning about components in the context of composition by using assume-guarantee reasoning [55].

Chapter 5

Implementation

This chapter presents the algorithm for checking composition, the implementation of activations and heaps, and I/O facilities for an interpreter designed for the rc$_{\text{go}}$programming language presented in Chapter 4.

5.1 Interpreter Organization and Implementation

The interpreter consists of a scanner, a parser, a sequence of semantic checks, a code generation phase, composition checks, and an execution phase. The interpreter is implemented in C++. Flex and Bison were used to implement the scanner and parser, respectively. The semantic checks include type checking and enforcement of intrinsic and indirection mutability as set forth in Chapter 4. The code generation phase converts the AST into a tree of stack operations. The composition check synthesizes transactions and checks them for soundness (Section 5.2). The execution phase begins by initializing component instances by calling the initializer associated with each instance. The scheduler then proceeds by executing transactions according to the fairness criteria of Chapter 6. The scheduler implementations use the POSIX threads (pthreads) library for concurrent execution and synchronization. The code is available on GitHub[6].

[6]https://github.com/jrw972/rcgo

5.2 Enforcing Sound Composition

This section outlines the algorithm for checking the composition semantics of reactive components. As described in Chapter 3, one of the goals for the reactive component model is to facilitate the construction of complex reactive systems through composition. The composition semantics of reactive components overcome limitations of I/O Automata and UNITY but introduce concurrency hazards that could result in systems whose behavior is not well defined. Two key hazards to be avoided are non-deterministic transactions arising from the same state being updated by multiple transitions within a transaction, and recursive transactions arising from cycles in composition. The goal of the composition check is to determine whether a system is free of these hazards.

Checking for sound composition is a holistic problem. Ports facilitate third-party composition by being opaque, meaning that the action or reaction activating a port cannot know about the state transitions executed as a result of activating the port. The state involved in a transaction is not known until the components are instantiated and the ports bound. Thus, checking for sound composition requires a reasonably complete understanding of the system. To this end, the algorithm described in this section leverages the static system assumption[7]. We leave relaxing the static system assumption, which will require extending the model and proposed checking algorithm, to future work.

The reactive component semantics introduced Chapter 3 entail a number of assumptions that allow composition checking to be formulated as a set of simple graph- and set-theoretic problems. Activate statements are limited to the bodies of actions and reactions, and port calls are limited to activate statements. Activate statements terminate the execution of an action or reaction, meaning at most one activate statement will be called per action or reaction body. Thus, a transaction can be viewed as a directed graph where each node is an action or reaction and edges indicate that the source action or reaction activates the target reaction. With this graph in place, the state involved in a transaction can be deduced by treating the component instances as proxies for their state variables by creating sets of instances and evaluating the sets for compatibility.

[7]I.e., that all components are known *a priori* and no components are added or removed from the system at runtime.

The checking algorithm consists of several distinct steps, each of which is described next in further detail. These steps are performed in the order they are presented as later steps depend on earlier ones.

Enumerate instances and ports. The first step to checking composition is to enumerate the components in the system using the static system assumption. The top-level components are given by the declared instances. Sub-components are enumerated by recursively instantiating fields that are also components. Let I denote the set of component instances. Fields that are ports are also enumerated. Let S denote the set of push ports and L denote the set of pull ports.

Enumerate bindings. Let i be a component instance of type c. Associated with c is a set of binders B. Each binder $b \in B$ is evaluated for i to create a set of bindings. A binding either binds a push port to a reaction or a getter to a pull port. Let R denote the set of reactions and G denote the set of getters. The result of enumerating the bindings is two functions (look-up tables). The function $\mathit{reactions} : S \rightarrow \{R\}$ maps a push port to a set of reactions. The function $\mathit{getters} : L \rightarrow \{G\}$ maps a pull port to a set of getters.

Check bindings. Inverting *reactions* yields a function that maps a reaction to a set of push ports, $\mathit{reactions}^{-1} : R \rightarrow \{S\}$. The $\mathit{reactions}^{-1}$ function is used to ensure that a reaction either is bound to one push port or is not bound:

$$\forall r \in R : |\mathit{reactions}^{-1}(r)| \leq 1 \tag{5.1}$$

The *getters* function is used to ensure that a pull port is bound to exactly one getter:

$$\forall l \in L : |\mathit{getters}(l)| = 1 \tag{5.2}$$

Enumerate transactions. Let i be a component instance of type c. Associated with c is a set of actions A. Each action $a \in A$ is evaluated for i to create a transaction. A *transaction* is a directed graph constructed as follows, as the example in Figure 5.1 illustrates:

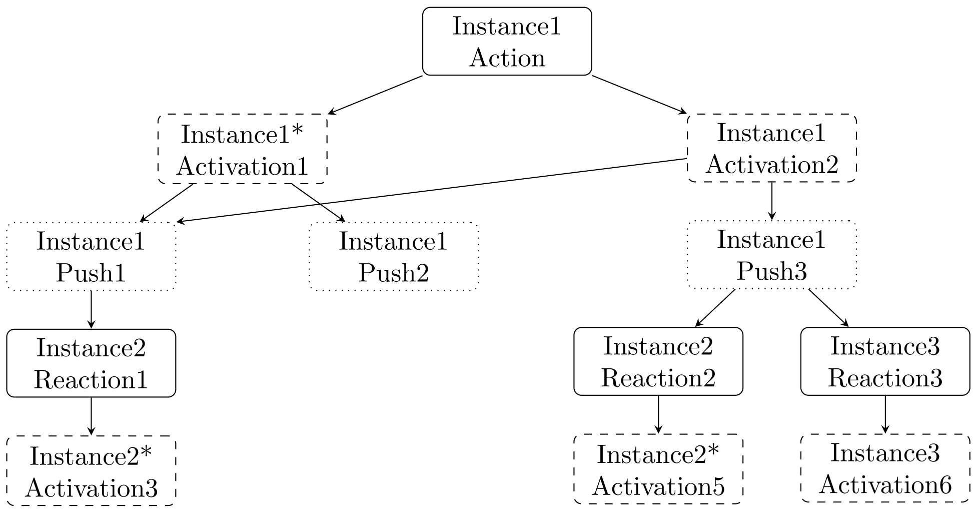

Figure 5.1: Example transaction

1. The root is an action a.
2. The descendants of the root are the activations in a and the pull ports and the getters used in the immutable phase.
3. The descendants of the pull ports are the getters bound to the pull ports given by *getters*.
4. The descendants of the activations are the push ports named in each activation.
5. The descendants of the push ports are the reactions bound to the push ports given by *reactions*.
6. This process is then repeated for each reaction and getter.

A recursive transaction is formed when a reaction activates itself through the set of bindings or a getter calls itself, which appears as a cycle in the transaction graph. The $\mathrm{rc}_{\mathrm{go}}$ interpreter uses an implementation of Tarjan's algorithm [90] to detect cycles.

A non-deterministic transaction occurs when the same state is manipulated by multiple transitions. Thus, the interpreter must determine what state is manipulated in a transaction to determine if the constituent transitions are compatible. The interpreter uses a component instance as a proxy for its state variables and determines how the state is accessed in each transition. The possible access patterns include:

Write in which at least one state variable may be mutated (* in Figure 5.1);

Read in which at least one state variable is accessed but no state variables are mutated; and

None in which no state variables are accessed.

The current implementation uses a conservative static analysis of the body of an activate statement to determine if the activation mutates the state of the component. For composition analysis, state variable access need only be determined for the mutable phase. However, performing a similar analysis for the immutable phase and precondition provides a complete description of state access in a transaction, which is used by multi-threaded schedulers to determine which transactions can be executed in parallel.

The check for non-deterministic transactions continues by confirming that all possible executions of a transaction are deterministic in two steps. First, two *access sets* are computed for each node in the transaction. The first access set describes what state is accessed in the immutable phase, while the second set describes what state is accessed in the mutable phase. Let $W = \{\mathit{Read}, \mathit{Write}\}$ be the set of relevant access patterns[8]. Each element in an access set $h \in H$ is a pair (i, w) where $i \in I$ and $w \in W$. Let $\mathit{inst} : A \cup R \cup G \to I$ be a function that maps an action, reaction, or getter to the corresponding instance. The immutable phase access sets are computed as follows:

- For an action, reaction, or getter denoted as x with instance $i = \mathit{inst}(x)$ that reads the state of i, the immutable phase access set is the union of the immutable phase access sets of its children and the set $\{(i, \mathit{Read})\}$.
- Otherwise, the immutable phase access set is just the union of the immutable phase access sets of its children.

The mutable phase access sets are computed as follows:

[8]The *None* access pattern is intentionally ignored as an optimization.

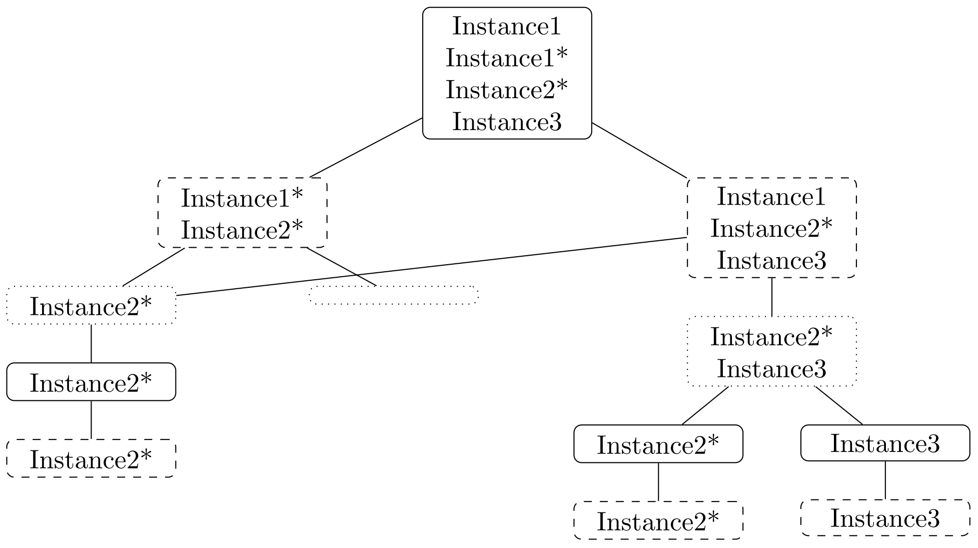

Figure 5.2: Example mutable phase access set calculation

- If the node is not an activation, or is an activation that does not access the state of the instance, then the mutable phase access set is the union of the mutable phase access sets of its children.
- Otherwise, the node is an activation belonging to instance $i \in I$ with access $w \in W$ and the mutable phase access set is the union of the mutable phase access sets of its children and $\{(i, w)\}$.

The access sets for the root of a transaction describe how state may be accessed during each phase of the transaction. An analysis similar to the immutable phase analysis may be applied to the precondition. All access pairs in the precondition and immutable phase access sets have *Read* access. Access pairs in the mutable phase access sets may either have *Read* or *Write* access. Activate statements that do nothing but log the state of a component are a common example of mutable phase access pairs with *Read* access.

Figure 5.2 shows the mutable phase access set calculation for the transaction illustrated in Figure 5.1. The root shows that Instance1 does not change in at least one activation (Instance1) and may change in at least one activation (Instance1*), that Instance2 may change in every activation (Instance2*), and that Instance3 is read-only in this transaction (Instance3). The empty node in Figure 5.2 comes from an unbound push port.

The second step for detecting a non-deterministic transaction is to verify that a mutated instance appears in at most one child node access set for the activation and push port nodes. Let $\mathit{race} : \{H\} \times \{H\} \to \mathcal{B}$ be a predicate that indicates a data race between two access sets. This function is defined as follows:

$$\begin{aligned}&\mathit{race}(H_1, H_2) = \\ &\qquad (\exists i : (i, \mathit{Write}) \in H_1 \wedge (i, x) \in H_2) \vee (\exists j : (j, \mathit{Write}) \in H_2 \wedge (j, y) \in H_1)\end{aligned} \tag{5.3}$$

This is, a component that changes state in one access set may not appear in the other access set. The *race* predicate is computed for each pair of child mutable phase access sets in activation and push port nodes. This check succeeds everywhere but Activation2 in Figures 5.1 and 5.2, as Instance2* appears in both children. The activation and push port nodes represent activities that will be performed together. That is, once control passes to an activate statement, all push ports and their bound reactions are activated. In contrast, activations (as children of action and reaction nodes) represent mutually exclusive alternatives: at most one activate statement is executed per action/reaction body. Thus, a mutated instance appearing in two or more children of an activation node or push port node indicates that the state of a component may be mutated in disparate ways leading to a non-deterministic transaction.

Complexity. Maintaining appropriate forward and reverse hash maps of the bindings allows the binding check to be performed in $O(N)$ time where N is the number of ports in the system. Similarly, the construction of a transaction graph can be performed in $O(|N| + |V|)$ time where $|N|$ is the number of nodes in the transaction and $|V|$ is the number of edges. Proving that a transaction is acyclic can be performed in $O(|N| + |V|)$ time where $|N|$ is the number of nodes and $|V|$ is the number of edges.

A loose upper-bound on the complexity of the access set calculations for the non-determinism check is $O(k|N|^2 \log(|N|))$ where $|N|$ represents the number of nodes in a transaction and k represents the maximum branching factor in the transaction. The size of the access set for the root is $|N|$. Assuming a naive set implementation, the complexity of computing the access set for the root is $O(|N| \log(|N|))$. This must be repeated k times for all nodes in the graph resulting in an overall complexity of $O(k|N|^2 \log(|N|))$.

A loose upper-bound on the complexity of the compatibility check is also $O(k|N|^2 \log(|N|))$. Assume that the size of the access set at each node is $|N|$. A tree-based set lookup can be performed in $O(\log(|N|))$ time. The lookup must be performed $k|N|$ times by the parent. The lookup must repeated for each of the $|N|$ nodes for a combined complexity of $O(k|N|^2 \log(|N|))$.

The complexity of these algorithms has not presented a problem in practice as most of the systems we have implemented have small numbers for $|N|$, $|V|$, and k.

5.3 Activations

When executing a transaction, the $\mathrm{rc}_{\mathrm{go}}$ run-time system must execute the immutable phase of all implied state transitions before executing any of the mutable phases. To accomplish this, the run-time system uses a novel calling convention to create a list of deferred contexts and statements that represent the mutable phase of each state transition. The immutable phase constructs the list and the mutable phase processes the list. To present the calling convention, we first present some details about the run-time system such as the ordinary calling convention and push ports. We then describe the behavior of the calling convention and explain it using an illustrative example.

Ordinary calling convention. The ordinary call mechanism in the $\mathrm{rc}_{\mathrm{go}}$ run-time system is similar to the C-decl calling convention. It assumes the existence of an *instruction pointer*, which contains the address of the currently executing instruction, and a *base pointer* that points to a location in the stack, which can be offset to access arguments and local variables. An ordinary call in the language is accomplished through the following sequence:

1. (Caller) Create space for return values.
2. (Caller) Push arguments onto the stack, left to right.
3. (Caller) Push the instruction pointer onto the stack and transfer control to the body of the function, method, action, reaction, getter, or initializer.

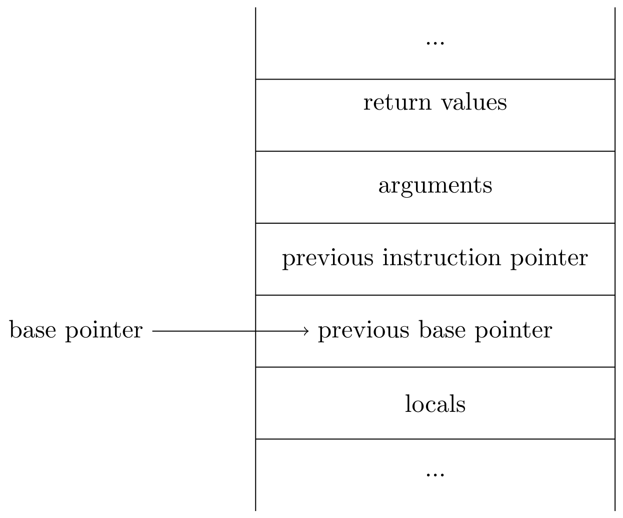

Figure 5.3: Diagram of a stack frame. The stack is depicted as growing down.

4. (Callee) Push the base pointer onto the stack and set the base pointer to the top of the stack.
5. (Callee) Reserve space on the stack for local variables.
6. (Callee) Execute the body of the function, method, etc.
7. (Callee) Pop the local variables, pop and restore the base pointer, pop and restore the instruction pointer.
8. (Caller) Pop the arguments.
9. (Caller) Pop the return values.

The major difference between this calling convention and C-decl is that the arguments are pushed in the opposite order to match the semantics of Go. The collection of arguments, previous instruction pointer, previous base pointer, and reserved space is called a *stack frame* (or *call frame*). Figure 5.3 shows the layout of a normal stack frame. For this presentation, we assume that the stack grows down, i.e., the previous base pointer has a lower address in memory than the previous instruction pointer.

Push ports. A push port is a field in a component, which is implemented as a pointer to a linked list that contains the component pointer/reaction pairs that are bound to the push port. The rc_{go} run-time system populates each push port before execution begins.

Synchronized two-phase calling convention. As was previously stated, the execution of an activate statement is split into an immutable phase and a mutable phase. To prepare for the mutable phase, the immutable phase must preserve the stack frame (context) of the action or reaction that executes an activate statement and must record which activate statement was executed so that the same activation can be resumed in the mutable phase. To accomplish this, we devised the *synchronized two-phase calling convention*, which is used to execute activate statements during the immutable phase.

Recall that activate statements may only appear in actions and reactions. After executing the immutable phase of an activate statement in a reaction, control must be returned to the calling activate statement so that it may activate other push ports. After executing the immutable phase of an activate statement in an action, control must be returned to the rc_{go} run-time system to begin the mutable phase.

In the ordinary calling convention, the stack frame for the reaction would be popped, control would be returned to the caller, and the arguments would be popped. To preserve the stack frame for use during the mutable phase, however, the activate statement returns control to the caller without popping the frame and the caller does not pop the arguments. Thus, the complete frame for the reaction is preserved on the stack.

Each such deferred stack frame is added to a list to make it available in the mutable phase. Let *head* be a variable containing a pointer, initially nil, which will serve as the head of a linked list. Before the activate statement returns from the immutable phase, it sets the previous base pointer in the deferred stack frame to the value of *head* and updates head to be the current base pointer which inserts the stack frame into the list. The rc_{go} run-time system can iterate over the elements of the list by following the previous base pointer to access all of the stack frames that are needed for the mutable phase. If the list is empty (*head* is nil), then no activate statement was executed and the mutable phase may be skipped.

The final piece of information that must be recorded is the body of the activate statement so that it is accessible in the mutable phase. Before returning from the immutable phase, the activate statement records the body of the activate statement in the previous instruction pointer slot of the current stack frame. It is safe to use the previous instruction pointer because it is not used beyond the immediate return. Furthermore, it is at a fixed location, which allows it to be accessed in any deferred stack frame.

The synchronized two-phase calling convention is used when executing an activate statement and proceeds as follows:

1. Save the previous base pointer of the current stack frame in *bp*.
2. Set the previous base pointer of the current stack frame to the value of *head*.
3. Set *head* to the value of the base pointer.
4. Save the previous instruction pointer of the current stack frame in *ip*.
5. Set the previous instruction pointer of the current stack frame to the body of the activate statement.
6. For each push port in the port call list:
    (a) Push the arguments to the push port onto the stack.
    (b) For each component pointer/reaction pair in the push port:
        i. Push the component pointer onto the stack.
        ii. Copy the arguments prepared in 6a onto the stack.
        iii. Call the reaction.
7. Set the base pointer to the value of *bp* and return control to the address in *ip*.

The synchronized two-phase calling convention evaluates the arguments to a push port once and passes a copy to each bound reaction. An alternative would be to evaluate the arguments for each bound reaction. This means that the arguments may not be evaluated (i.e., the push port is not bound to any reactions) or may be evaluated multiple times. If the arguments contain an expression with side-effects, then the behavior of the code becomes dependent on composition. While this may be desirable in some cases, we opted for making a port call resemble an ordinary function call as much as possible. This sentiment also influenced our decision to pass a copy of the arguments to each reaction as opposed to reusing the same set of prepared arguments. Since each reaction starts with a copy of the arguments, it is free to manipulate them as allowed by the semantics of Go. Furthermore, the arguments are available in the mutable phase which obviates the need to make local copies of arguments. This approach assumes that copying arguments does not generate significant overhead.

A major caveat when using the synchronized two-phase calling convention is that it must be assumed that the stack pointer changes when calling a push port. To illustrate why this is an issue, consider the case when a caller wishes to preserve the contents of a register. One strategy is to push the contents of the register onto the stack, perform the call, and then pop the value from the stack into the register. After calling a push port, the caller may no longer assume that the previous value of the register is at the top of the stack. An easy work-around is to allocate local variables, i.e., variables whose addresses are relative to the base pointer instead of the stack pointer, for saving temporary values that must persist across a push port call. In the same way, the variables used to iterate over the list of component/reaction pairs in a push port should be allocated as local variables so that they may survive the calls to the reactions.

Mutable phase. The mutable phase consists of executing all of the deferred activate statements. The algorithm for doing so is as follows:

1. If the value of *head* is nil, stop. Otherwise, set the base pointer to the value in *head*.
2. Transfer control to the instruction indicated by the previous instruction pointer. Execution continues until the body of the activate statement (implicitly or explicitly) returns.
3. If the previous base pointer is nil, stop. Otherwise, set the base pointer to the previous base pointer and go to 2.

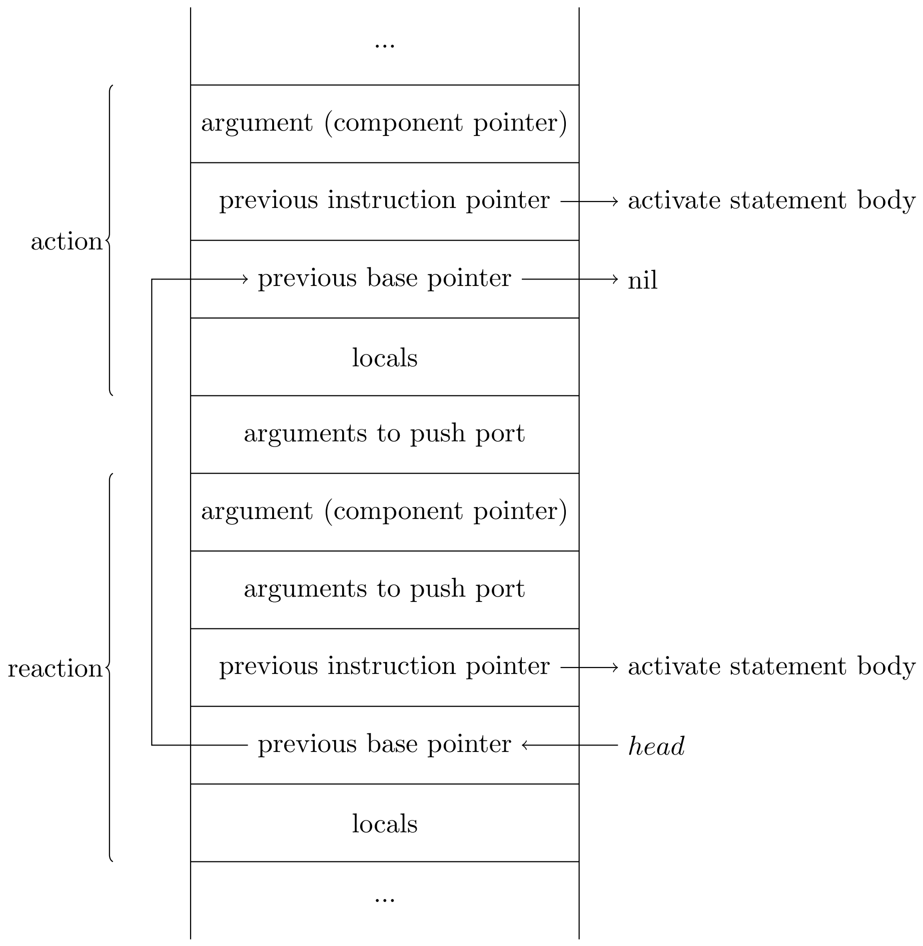

Figure 5.4: Diagram of the stack after the immutable phase when an action activates a single reaction

The algorithm iterates over the list of deferred stack frames accessible through *head*. The last element in the list is indicated by a previous base pointer that is nil. Control is transferred to the previous instruction pointer which contains the body of the activate statement.

Example: one action, one reaction. Suppose an action activates a single push port that is bound to a single reaction and the reaction has a single activation that does not activate any push ports. Figure 5.4 shows a diagram of the stack after the immutable phase for this scenario. The stack contains two frames, one corresponding to the action and one corresponding to the reaction. The *head* variable points to the reaction frame which in turn points to the action frame using the previous base pointer slot. The action frame points to nil indicating that it is the last frame in the list. The previous instruction pointers point to the bodies of the activate statements. Between the frames are the push port arguments which are duplicated for the call to the reaction. If the push port had been bound to multiple reactions, then the reaction portion of the diagram would be replicated to match the number of bound reactions. If multiple push ports were activated by the activate statement, then additional push port arguments and reactions would appear on the stack.

Calling convention efficiency. The synchronized two-phase calling convention has the potential to be as efficient as a function or method call. The proposed calling convention can be implemented directly on modern hardware architectures like x86 and x86_64, which contain all of the registers and instructions necessary to support the synchronized two-phase calling convention. The synchronized two-phase calling convention relies solely on the stack which means that the underlying operating system must set up the stack, back it with memory pages, etc. More importantly, it avoids the overhead of heap allocation. Port calls resemble virtual method calls in that the reactions may need to be looked up before they can be executed. However, this lookup could be avoided by inlining the body of each reaction using the substitutional equivalence property. This could be performed prior to execution or during execution using just-in-time compilation techniques. We leave the application of these techniques to future work.

5.4 Heaps

In this section, we describe the implementation of the heap data type introduced in Chapter 4 and our approach to garbage collection. Our approach to implementing dynamic memory allocation and garbage collection was shaped by the independence of state required by the semantics of reactive components. Thus, instead of relying on a single global heap, the implementation contains a heap for each component that can be garbage collected independently of the others. This independence allows garbage collection to be performed concurrently with other activities using simple, single-threaded algorithms.

Slots and blocks. A *slot* is the smallest unit of memory that can be dynamically allocated. Typically, a slot is the size of two pointers. A *block* is an extent of memory and a set of bits indicating the allocation status of each slot in the extent. The status bits indicate if a slot is allocated, if a slot is the beginning of an object, and if the slot has been marked by the mark-and-sweep algorithm. Blocks also contain left and right pointers that allow them to be formed into a binary tree ordered by the address of the extent.

Mark-and-sweep garbage collection. We implemented a simple mark-and-sweep algorithm to collect garbage in a tree of blocks. The core of the marking phase involves scanning extents for pointers to objects. When the algorithm comes across a slot that is allocated, marked, and which contains a pointer-sized value that points to a location in the tree of blocks that is also allocated, it marks the object indicated by the pointer. The algorithm is bootstrapped by marking all slots in a designated root object. The algorithm repeatedly scans the extents in the tree of blocks until no new marks can be added. At this point, the sweep phase resets the allocated bit for all slots that are allocated but not marked and resets the marked bit for the next run of the algorithm. A block that is not marked, meaning that none of its slots were marked, is removed from the tree of blocks and deallocated.

Heaps. A *heap* contains a root block, a list of unallocated chunks of memory (free list), and pointers to create a tree structure. When allocating an object from a heap, the heap attempts to find an adequate chunk in the free list using a first-fit policy. If this fails, the heap allocates a new block, inserts it into the tree and free list, and then allocates again using the newly inserted chunk. The sweep phase of the mark-and-sweep algorithm reconstructs the free list.

As described in Chapter 4, every component has an associated heap. A heap of this kind is called an *implicit heap*. Heaps that are created via `new` and passed to `move`, `merge`, and `change` are called *explicit heaps*. All heaps have a distinguished root object. The marking phase of the mark-and-sweep algorithm is seeded with this root object. The root object of an implicit heap contains the statically allocated state of the component that owns the heap. The root object of an explicit heap is an object that is allocated in the same heap.

The semantics of reactive components allow heaps to form strict hierarchies, i.e., a tree structure where the root of the tree is an implicit heap and the internal nodes and leaves of the tree are explicit heaps. A strict hierarchy gives a graphical interpretation to merge and move operations. A move operation moves a sub-tree from one location to another location. Merging a heap h into another heap p involves removing h from the tree, inserting the blocks of h into the tree of blocks of p, merging the free list of h into the free list of p, and inserting the children of h as children of p.

The active heap. Logically, the rc_{go} run-time system maintains a stack of heaps where the top of the stack represents the *active heap*. The active heap is used to satisfy all dynamic memory requests, i.e., calls to `new`. The implicit heap that is associated with the receiving component is pushed/popped upon entering/leaving an action, reaction, getter, or initializer. Specific `change` statements are used to push and pop explicit heaps. Thus, the call stack (i.e., the `change` statements that are active on the call stack) implements the stack of heaps. When a new heap is created, it is inserted as a child of the active heap.

Atomicity. Chapter 4 describes how the semantics of reactive components permit concurrent access to heaps. The first scenario where this may occur is when a heap is passed to a push port that is bound to multiple reactions. The semantics of reactive components allow the reactions to be executed concurrently. Thus, two different threads may attempt to move/merge the same heap at the same time. The second scenario occurs when a component is concurrently accessed in multiple transactions that don't mutate the state of the component. The action, reaction, or getter is allowed to allocate memory, which means that heaps must correctly handle concurrent access. Our implementation of heaps uses the Thread Safe Interface pattern [84] to synchronize access to heaps when allocating, moving, and merging.

Heap links. Heaps are exposed to users via pointers to heaps, e.g., `*heap int`. These pointers to heaps can be stored in objects allocated in another heap. The semantics of `change`, `merge`, and `move` ensure that the parent-child relationships formed by pointers to heaps match exactly those known by the run-time system. The two rules that enforce this behavior are that 1) `merge` and `move` fail for any heap that is already on the stack of active heaps and 2) all pointers in scope become foreign in the body of a change statement. The

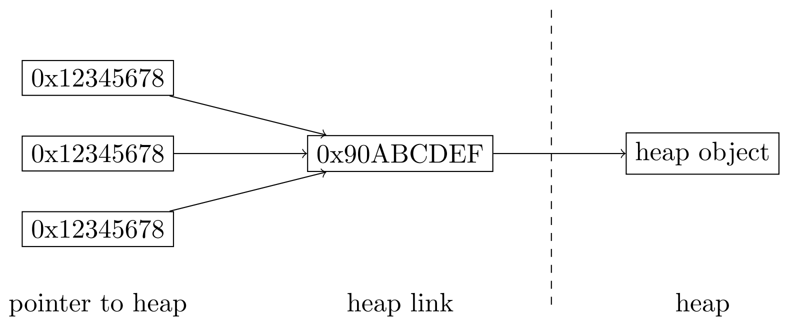

Figure 5.5: Example illustrating heap links. The objects on the far left represent user-level pointers to heaps, e.g., `* heap int`. The middle object is a heap link, which contains the address of the child heap that appears on the far right.

second rule forces the user to `move` heaps if they desire to change the hierarchy, because pointers with foreign indirection mutability cannot be stored. Recall that the stack of active heaps is set up by active `change` statements. Using the inductive argument that the parent-child relationship was previously correct, then the ancestors of the active heap must all appear in the stack of active heaps. Thus, failing to `merge` and `move` heaps on the active stack prevents the formation of cycles.

As illustrated in Figure 5.5, a user-level pointer to a heap points to an object called a heap link which in turn points to a heap. The objects on the left side of the diagram are objects in the parent heap and the object on the right is the child heap. The extra level of indirection introduced by the heap link concentrates all references to the child heap into one location. Moving a heap involves reading the pointer out of the heap link and replacing it with `nil`. This makes the heap inaccessible to the previous owner (parent), which maintains the isolation of state between reactive components. The marking phase of the mark-and-sweep algorithm is aware of heap links and marks heaps as being reachable from another heap. Unreachable heaps are pruned from the hierarchy during garbage collection.

Concurrent garbage collection. Two features of reactive components permit concurrent garbage collection. First, the state of each reactive component is isolated which allows garbage collection to be performed on different components at the same time without conflict. Second, garbage collection can be framed as an action to be executed by the scheduler. Thus, associated with each component is an action that invokes the garbage collector on the implicit and explicit heaps of the corresponding component. Furthermore, the action is considered to modify the state of the component which prevents all transactions involving the component from executing concurrently with the garbage collection action. We assume that the scheduler executes the garbage collection action often enough that no additional invocations of the garbage collector are necessary. More plainly, we do not trigger the garbage collector in the allocation routine, which is the approach taken by many systems using garbage collection. Most garbage collection algorithms include a scan of the call stack as objects reachable from the call stack must not be collected. In our approach, such scanning is useless since the call stack is empty when executing the garbage collection action and the root object is embedded in the implicit heap that is being collected.

5.5 I/O

A possible bottom-up approach to support reactive components would be to write a kernel for reactive components, i.e., a scheduler and memory manager, and then write an operating system and application suites using reactive components. This approach, however, is impractical and risky given the expense of developing software for an as yet unproven technology. A top-down approach instead involves proving the utility of reactive components at the application layer and then proceeding to lower layers when appropriate. This approach far less risky and is the one taken in this chapter. To develop and evaluate real-world applications, the input/output facilities of the host operating system then must be exposed to reactive components so that they may communicate and interact with other processes and systems. This section describes our approach to exposing the input/output (I/O) facilities of a Linux/GNU system to reactive components.

A Linux/GNU system offers a variety of I/O and communication mechanisms including pipes, sockets, shared memory, and message queues. For our purposes, we focused on mechanisms that are available through a file descriptor interface. Our approach consists of two steps. The first step is to add file descriptors and operations for manipulating them, such as read and write, to the $\mathrm{rc}_{\mathrm{go}}$ language. The second step was to wrap file descriptors in reactive components to make their functionality available via a conventional reactive component interface.

The main consideration when supporting file descriptors was to ensure that reads and writes were non-blocking, as a blocking read or write may violate fairness. A transaction that blocks on a read or write would adversely affect the latency and throughput of the scheduler as the scheduler thread would be blocked and not able to service other transactions. Furthermore, a blocking transaction holds the locks for all components involved in that transaction, and other enabled transactions that also involve those components would be denied service which also violates the fairness requirement of the scheduler. To prevent these problems, all file descriptors when they are created are set to be non-blocking. Thus, all subsequent reads and writes are also non-blocking.

Threaded programs using non-blocking I/O typically use some kind of synchronous I/O multiplexing which allows a thread to determine which file descriptors are ready for reading and/or writing. The goal in doing so is to allow a thread to service multiple file descriptors and yield the processor if no file descriptors are ready. To map this concept into reactive components, we introduced a `readable` function that tests if a file descriptor is ready for reading and a `writable` function that tests if a file descriptor is ready for writing. These functions are intended to be used in preconditions so that an action becomes enabled when the corresponding file descriptor is ready.

The `readable` and `writable` functions are also used as part of the termination protocol of the Partitioned scheduler described in Section 6.4. When the termination protocol enters the checking phase where it attempts to prove that every precondition is false, it records which file descriptors are being checking for readability and writability. If the termination protocol proves that all preconditions are false but some preconditions depend on readability or writability, it enters a state where it waits for one of the file descriptors to become ready. This allows a system of reactive components to sleep while it waits for external input like a message from a remote host or a timer.

Conceptually, a file descriptor contains component state and should be subject to all of the constraints of normal component state. The two main constraints are 1) it cannot be shared by another component and 2) it cannot change in the immutable phase of a transaction. To enforce the first constraint, file descriptors are implemented as dynamically allocated opaque data structures. Forcing dynamic allocation makes file descriptors subject to the pointer sharing rules, i.e., they can only be shared via foreign indirection mutability in the

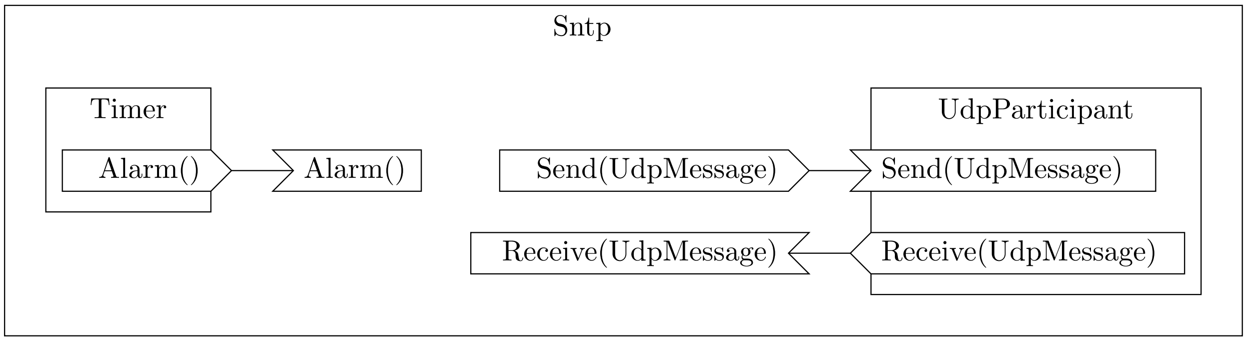

Figure 5.6: Diagram of a Simple Network Time Protocol (SNTP) client

immutable phase. Making the data structure opaque prevents sharing through copying. The normal immutable phase rules in concert with properly declared built-in functions prevent file descriptors from changing in the immutable phase. That is, functions like `read` and `write` require a pointer to a file descriptor with mutable indirection mutability.

To test these ideas, we implemented a Simple Network Time Protocol (SNTP) client. SNTP attempts to acquire the current time from a server while simultaneously measuring a round-trip latency to determine an accurate local time. The client timestamps the request with the local time and sends it to the server. The server timestamps the request when it arrives and again when it sends it back to the client. Finally, the client timestamps the request when it receives it back from server. The client uses the timestamps to determine the offset of the local clock from the clock on the server. This procedure is repeated periodically to synchronize the local clock. The messages are exchanged using the User Datagram Protocol (UDP).

Figure 5.6 shows a diagram of the SNTP client application. The top-level Sntp component contains two sub-components: Timer and UdpParticipant. The Timer component is a periodic timer implemented by wrapping a `timerfd` file descriptor. The UdpParticipant component wraps a UDP socket file descriptor and is capable of sending and receiving UDP messages. The Sntp component uses these components to implement an SNTP client. Whenever the Timer fires the alarm, the UdpParticipant creates an SNTP request and sends it using the UdpParticipant. This all happens as part of one atomic transaction. Whenever the UdpParticipant receives a message, it passes it to the Sntp component which deserializes it, timestamps it, and interprets it to compute the local clock offset.

The Sntp component demonstrates how composition (with reactive components for I/O) can be used to construct systems. The Timer and UdpParticipant components are generic since they do not contain any SNTP related logic. Furthermore, they have well-defined concurrency semantics that will be enforced when they are composed in other systems. Thus, it is possible to use the reactive component model to produce reusable *reactive* software.

5.6 Summary

The goal of implementing reactive components was to test the practicality of the reactive component model. Specifically, we desired to know how the various features of the model could be implemented, which features were troublesome, and how the implementation utilizes various assumptions. Flexibility in the reactive component model necessitates a check for sound composition. The sound composition algorithm described in Section 5.2 uses the static system assumption and the component proxy assumption to generate a graphical model of each transition and check it for determinism. A cursory analysis yielded an upper bound of $O(kn^2 \log(n))$ where n is the number of instances involved in the transaction and k is the maximum branching factor in the transaction.

To implement activate statements, we developed the synchronized two-phase calling convention which executes the immutable phase of a transaction first and preserves the context necessary for the second (mutable) phase. The synchronized two-phase calling convention was implemented using standard function call and stack manipulation facilities.

The isolation of component state is enforced in two ways. First, the type system prevents components from sharing pointers directly via `$const` and `$foreign` modifiers. Second, the implementation uses garbage collection to prevent indirect sharing through dangling pointers. In this chapter, we describe the implementation of heaps and how they are used for dynamic memory allocation. Heaps are designed so that they can be merged and moved. At the user level, all references to a heap are encapsulated by a “link” which is used to enforce atomic moves and merges. The independence of state between components allows each component to have its own heap (or tree of heaps) that can be garbage collected independently of other components. Garbage collection can be conducted in parallel by associating a garbage collection action with each component.

To enhance the practicality of the reactive component implementation, we introduced file descriptor I/O that allows reactive components to interact with a variety of operating system facilities. All I/O is non-blocking to preserve the fairness of the scheduler. Components may use the `readable` and `writable` functions to test file descriptors. The scheduler uses these functions as part of the termination protocol to sleep while waiting for external inputs. File descriptors are treated as component state and therefore cannot be shared between components or mutated during the immutable phase. Using file descriptor I/O, we implemented an SNTP client, which demonstrates how systems can be built up from simple components.

Chapter 6

Transaction Scheduling

A scheduler is responsible for executing transactions according to the fairness criteria set forth in Section 3.1.5. Of particular interest is the design and implementation of general-purpose schedulers that are capable of executing transactions in parallel. The main inputs to a scheduler are the transactions enumerated by the composition analysis. The access sets calculated for each transaction are used by concurrent schedulers to avoid non-deterministic state transitions. In this chapter, we introduce some of the issues involved in designing and implementing a scheduler for reactive components, through a series of increasingly complex scheduler designs. The two concrete implementations described later will draw upon the designs of these schedulers. An evaluation of these schedulers indicates that reactive components may be a viable event-based alternative to threads in terms of performance but additional work is necessary to generalize this claim.

6.1 The Transaction Scheduling Problem

A scheduler is responsible for mapping transactions to processor cores so that they can be executed according to the semantics of reactive components. As described in Section 3.1.5, concurrent execution is modeled by a scheduler that serially and repeatedly executes atomic transactions according to fairness, which means that all enabled transaction must eventually be executed. A scheduler implementation supporting concurrent execution must take care to preserve the fairness and atomicity required by the model. A scheduler is *fair* if it executes each transaction an infinite number of times or terminates by reaching a fixed point. A scheduler is *safe* if it avoids conditions where one transaction is changing the state of a component while another transaction is reading or writing the same state. The execution of a transaction in a safe scheduler is logically atomic.

A scheduler is *responsible* if it only terminates when a fixed point has been reached. Termination is not a requirement for a scheduler, but it is useful from the perspective of testing and evaluation. We are interested in designing fair, safe, and responsible schedulers, as these schedulers enforce the semantics of reactive components.

We model a scheduler as a state transition system consisting of a set of possible states S, an initial state $s_0 \in S$, and a transition function $\sigma : S \rightarrow S$. The function σ is applied to the current scheduler state s to generate the next scheduler state $s' = \sigma(s)$. To refer to previous scheduler states, we assign a logical time $t \in \mathcal{N}$ to each scheduler state so that $s(t+1) = \sigma(s(t))$. Using this notation, the initial state of the scheduler is $s(0) = s_0$. The definitions of S, s_0, and σ will be unique to each scheduling algorithm. We are interested in the properties of σ as they relate to enforcing the scheduling semantics of reactive components and how these properties help or hinder a scheduler implementation.

The generic state of a scheduler is modeled as a vector where each element of the vector contains the scheduling state associated with a particular transaction. Let $s \in S$ be a generic scheduler state and let T be the set of transactions. The value $s_u(t)$ represents the scheduler state for transaction $u \in T$ at time t. Each value $s_u(t)$ is a pair (p, q). The value $p \in \{\bot, 0, 1\}$ represents the precondition of the transaction known by the scheduler as either unknown $(\bot)$, false (0), or true (1). The value $q \in Q = \{\mathit{Idle}, \mathit{Eval}, \mathit{Exec}\}$ represents the state of the transaction as either being idle, evaluating the precondition, or executing the immutable and then mutable phases. Initially, all transactions start with an unknown precondition in the idle state:

$$\forall u \in T : s_u(0) = (\bot, \mathit{Idle}) \tag{6.1}$$

In the model, the precondition of each transaction is always defined and known since the state against which the preconditions are evaluated is always defined and known due to atomicity. In a real scheduler, the execution of a transaction takes time and the state of all involved components is not defined for this duration. Consequently, the values of preconditions derived from this state are also not defined during the same duration. After

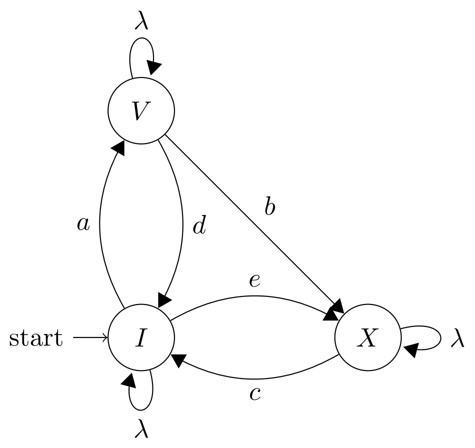

Figure 6.1: State transition diagram for the state component of the dynamic transaction state. *I* represents *Idle*, *V* represents *Eval*, and *X* represents *Exec*.

a transaction, all preconditions based on any mutated state are undefined and must be re-evaluated to determine if they are true or false.

The dynamic transaction state $s_u(t)$ is itself a state transition system and much of the challenge in designing a scheduler revolves around the details of enforcing this transition system for concurrently executing transactions. Figure 6.1 shows the state transition diagram for the state component of the dynamic transaction state (q of $s_u(t)$). Each state has a self loop that allows a transaction to stay in the same state while another transaction changes state (indicated by λ). There are two main paths through the transition system. The path *abc* corresponds to evaluating the precondition and immediately executing the immutable and mutable phases. The path *adec* corresponds to evaluating the precondition and later executing the immutable and mutable phases. Let Γ be the set of transitions indicated in Figure 6.1. Each element $\gamma \in \Gamma$ is a pair in $Q \times Q$. The following condition states that every transition in a scheduler must obey the transition system described in Figure 6.1:

$$\forall t, u : (s_u(t).q, s_u(t+1).q) \in \Gamma \tag{6.2}$$

The precondition of a transaction is established as true or false as a result of leaving the *Eval* state (transitions b and d in Figure 6.1):

$$\forall t, u, s_u(t).q = \mathit{Eval} \wedge s_u(t+1).q \neq \mathit{Eval} : s_u(t+1).p \in \{0, 1\} \tag{6.3}$$

The precondition of a transaction must be true while executing the immutable and mutable phases:

$$\forall t, u : s_u(t).q = \mathit{Exec} \implies s_u(t).p = 1 \tag{6.4}$$

A transaction that mutates the state of one or more components invalidates the precondition of all transactions whose preconditions are derived from the same state. Let $\mathit{pre} : T \to H$ be a function that maps a transaction to the set of access pairs corresponding to the precondition of a transaction. Let $\mathit{imm}(u)$ and $\mathit{mut}(u)$ be similarly defined functions that return the immutable phase access set and mutable phase access set for transaction $u \in T$. All of these functions may be computed as part of composition analysis. The set of transactions whose "enabledness" is potentially affected by a given transaction $u \in T$ is given by the following function[9]:

$$\mathit{affected}(u) = v \in T : \mathit{race}(\mathit{mut}(u), \mathit{pre}(v)) \tag{6.5}$$

The result of a transaction being executed has the effect of invalidating the precondition for all affected transactions:

$$\forall t, u : s_u(t).q = \mathit{Exec} \land s_u(t+1).q \neq \mathit{Exec} \implies \\ \forall v \in \mathit{affected}(u) : s_v(t+1).p = \bot \tag{6.6}$$

The previous requirement represents a worst case scenario where the transaction mutates all components in the mutable access set. An obvious optimization is to only invalidate preconditions derived from the subset of instances that actually changed state.

A scheduler is safe if it avoids data races among the set of active transactions. We define the dynamic access set of a transaction as follows:

$$\mathit{access}(s_u) = \begin{cases} \emptyset & \text{if } s_u.q = \mathit{Idle} \\ \mathit{pre}(u) & \text{if } s_u.q = \mathit{Eval} \\ \mathit{imm}(u) \cup \mathit{mut}(u) & \text{if } s_u.q = \mathit{Exec} \end{cases} \tag{6.7}$$

[9]The *race* function is defined in Section 5.2.

A scheduler state is safe if there are no data races:

$$safe(s) = \forall u, v \in T, u \neq v : \neg race(access(s_u), access(s_v)) \tag{6.8}$$

A scheduler is safe if it only enters safe states:

$$\forall t : safe(s(t)) \tag{6.9}$$

In a fair scheduler, an enabled transaction cannot be postponed indefinitely. Argument by contradiction is one approach to demonstrating that a scheduling algorithm is fair. If one assumes that a scheduler is not fair, then there must be some scheduler state $s(t_c)$ that contains a transaction u that is enabled in every subsequent state but never executed. The "enabledness" described in the previous sentence is not the value p in the definition of $s_u(t)$ which is the value known by the scheduler. Rather, it refers to the actual value of the precondition based on the state contained in the component instance. The argument is completed by demonstrating that the scheduler does in fact execute u. For example, a scheduler that processes transactions in first-in first-out (FIFO) order eventually executes every transaction in the queue.

A scheduler design must reconcile the forces arising from the basic execution model, fairness, safety, and efficiency. The basic execution model forces the scheduler to establish preconditions (Equation 6.3) before executing the immutable and mutable phases (Equation 6.4). Fairness compels the scheduler to execute certain transactions while safety compels the scheduler to avoid executing certain transactions (Equation 6.9). With respect to efficiency, a scheduler design may attempt to exploit the concurrency available in the set of transactions through parallel execution.

To illustrate the interplay among these forces, consider a concurrent and work-conserving scheduler. Let the system consist of the set of transactions $S \cup \{\tau\}$ where S is a set of transactions that are safe with respect to each other and τ is a transaction that is not safe with respect to all transactions in S. Thus, at any given time, the scheduler can be executing transactions in S or τ. Suppose the scheduler is concurrently executing two transactions from S. When one of the transactions terminates, the scheduler must immediately execute another transaction since it is work conserving. For safety, the scheduler will execute another transaction from S as it is unsafe to execute τ. This pattern of behavior induced by the work conserving nature of the scheduler may perpetually deny service to τ. For fairness, the scheduler must refrain from executing another transaction so that the τ transaction can be serviced. Thus, with respect to parallel and concurrent execution, a scheduler design must balance gains in performance from exploiting parallelism with the requirement of fairness.

The scheduling problem for reactive components, then, is: given a set of active transactions, select the next transaction for evaluation (precondition) or execution (immutable and mutable phase) subject to fairness and safety.

6.2 Scheduler Design Criteria

Different factors may influence the design of transaction schedulers, which we distinguish according to four dimensions. A given scheduler may be: *lazy* or *eager*, *oblivious* or *knowledgeable*, *cautious* or *speculative*, and *non-preemptive* or *preemptive.*

Lazy and eager schedulers. A scheduler may be *lazy* or *eager* with respect to evaluating preconditions. A lazy scheduler defers evaluating the precondition until the transaction is selected for execution (path *abc* in Figure 6.1). An eager scheduler evaluates preconditions after the state upon which they are based changes (path *ad* in Figure 6.1). The perceived benefit of a lazy scheduler is that it may reduce overhead by not evaluating preconditions while the perceived benefit of an eager scheduler is that it may only select actions which are enabled, which may result in more efficient use of acquired resources.

Oblivious and knowledgeable schedulers. We have considered two kinds of schedulers with respect to safety. The first is an *oblivious* scheduler that doesn't have direct access to the (global) scheduler state and therefore can't know the set of next states. An oblivious scheduler selects a transaction and then determines if the transaction is safe to execute. This typically involves a locking mechanism to ensure that all of the instances in the requisite access sets are available. The second kind of scheduler is a *knowledgeable* scheduler that does know the global scheduler state. A knowledgeable scheduler can be proactive and maintain a set of transactions that are safe with respect to the set of active transactions. This would allow a knowledgeable scheduler to efficiently select a safe action. The open question is whether or not the efficiency of selection overcomes the overhead of maintaining the set of safe transactions.

Cautious and speculative schedulers. A *cautious* scheduler avoids all race conditions and always satisfies the safety requirement of Equation 6.9. A *speculative* scheduler optimistically evaluates preconditions and executes transactions but aborts them if a conflict is discovered. Transactional memory is one technology that could be used to build a speculative scheduler. For this work, we will assume that preconditions, immutable phases, and mutable phases are not aborted and leave the application of transactional memory to reactive components as future work.

Non-preemptive and preemptive schedulers. An *active* transaction is one whose precondition is being evaluated or whose immutable or mutable phase is being executed. An active transaction need not physically occupy a processor core. This condition occurs in *preemptive* schedulers that can interrupt a precondition, immutable phase, or mutable phase to do other work. In contrast, a *non-preemptive* scheduler does not interrupt the execution of a transaction, i.e., transactions are physically atomic. Preemption may be useful to reduce the latency of transactions, support real-time priorities, etc. We leave the application of preemption to reactive component schedulers as future work.

6.3 Scheduler Design

The previous sub-section illustrated a number of design dimensions for reactive component schedulers: lazy vs. eager, oblivious vs. knowledgeable, cautious vs. speculative, and non-preemptive vs. preemptive. For tractability, we will focus on the design and implementation of particular schedulers that are lazy, oblivious, cautious, and non-preemptive, and leave a more complete exploration of the design space for future work. The goal of the following discussion is to introduce some of the issues when designing and implementing a scheduler, through a series of increasingly complex scheduler designs. We will describe the concurrent schedulers in terms of *threads*, which may be dedicated to physical processor cores or scheduled on one or more cores by an operating system.

Serial round-robin scheduler. Perhaps the simplest scheduler that can be implemented is a serial round-robin scheduler. This design may be appropriate in the context of uniprocessor embedded systems with tight resource constraints. The scheduler repeatedly cycles through the list of transactions, evaluating the precondition of each transaction and executing the immutable and mutable phases if the precondition was true. If no transaction is executed in a cycle, then the scheduler terminates. This scheduling algorithm is fair by virtue of the strict round-robin policy, safe from the fact that it is serial and non-preemptive, and responsible by definition. The *selection efficiency* of a scheduler is the number of transactions that must be selected before executing an enabled transaction or terminating. The selection efficiency of this algorithm is $O(|T|)$ as illustrated by a system where one transaction that is perpetually enabled while all of the other transactions are perpetually disabled.

Serial scheduler with a transaction work queue. The perceived weakness of the serial round-robin scheduler is the overhead of evaluating preconditions that evaluate to false. So, instead of cycling through the list of transactions, this scheduler maintains a work queue of transactions whose preconditions are true. At initialization time, the scheduler populates the queue by evaluating the precondition of each transaction in the system. The scheduler then repeatedly takes a transaction from the queue, executes it, and then inserts any newly enabled transactions, making this scheduler an eager scheduler. To find newly enabled transactions, the scheduler uses the access set generated during composition analysis. That is, after executing a transaction, the scheduler tests all of the transactions for the components in the access set and adds them to the queue if necessary. Alternatively, the scheduler may assume that the precondition is true and insert the transaction, knowing that the precondition will be re-evaluated when the transaction is selected. The scheduler terminates when the work queue is empty. This scheduling algorithm is fair if the work queue is processed in first-in first-out (FIFO) order, safe because the algorithm is serial and non-preemptive, and responsible since an empty work queue implies no transaction is enabled.

The pathological scenario for the serial round-robin scheduler applies to this scheduler as well, so the worst-case selection efficiency of the algorithm is $O(|T|)$. Assume that the most complex transaction in the system involves c components and some component has a system-wide maximum transaction count of a. In the worst case, the scheduler must evaluate $c \times a$ preconditions for every transaction that it executes. Thus, the overhead associated with this scheduler is related to the compositional structure of the system that it is executing. This overhead may be acceptable for systems where c and a are small or for systems whose average work queue size is small compared to the total number of transactions in the system. This suggests that the average number of enabled transactions compared to the total number of transactions may be a useful way to analyze reactive component systems and schedulers. For example, the serial round-robin scheduler would be appropriate for a *heavily enabled* system while the serial scheduler with a transaction work queue would be appropriate for a *lightly enabled* system.

Serial scheduler with an instance work queue. This scheduling algorithm attempts to squeeze a little more performance from the single-threaded scheduler with a transaction work queue. The significant difference is that after the scheduler executes a transaction, it places the component instances that might have changed state into the work queue. The work queue is initialized with all of the components in the system. When processing a component on the work queue, the scheduler cycles through all of the transactions for that particular component. The potential increase in performance comes from deferring the evaluations of the preconditions until absolutely necessary, i.e., lazy scheduling. The scenario on which this scheduler attempts to capitalize is when some components are rarely involved in transactions. An instance is enabled if at least one of its transactions is enabled. Thus, this scheduler is appropriate for systems that are lightly enabled from the instance perspective.

Concurrent global round-robin scheduler. This scheduler runs $P > 1$ copies of a round-robin scheduler in parallel. To be safe, we must devise a protocol that allows the threads to avoid concurrently executing transactions that may mutate the same state. Using the assumption that a component instance is a proxy for its state variables, the scheduler locks all instances in the access set before evaluating the precondition and executing the transaction. Recall that composition analysis determines the set of components that are involved in a transaction and how they are accessed (*Read* or *Write*). The acquired lock corresponds to the access type, which allows multiple threads to be reading a component but only one thread to be writing. The locks are acquired in a determined order to avoid deadlock (Havender's Principle). The concurrent global round-robin scheduler is fair so long as the underlying locking mechanism is fair, i.e., there is no reader or writer starvation. The algorithm is also responsible, as each thread proves to itself that there are no enabled transactions left in the system.

An important concern with this algorithm is the locking required to coordinate access to component instances. One negative characteristic caused by the locking is that a thread may become idle while waiting for a lock. Thus, we might look for alternatives that allow a thread to do other useful work while waiting for a lock. Another goal might be to look for ways to coordinate access to component instances without using locks at all. Some of these ideas are explored later in this chapter.

Concurrent partitioned round-robin scheduler. Rather than have each thread cycle through the entire list of transactions, the concurrent partitioned round-robin scheduler divides the list of transactions among the available threads. Like the global version, this algorithm is fair if the underlying locking mechanism is fair. Similarly, this algorithm is safe as locks are used to coordinate access to component state. However, for this algorithm to be responsible, we must add a protocol that allows the threads to detect that the system has reached the termination condition.

The termination protocol consists of a barrier synchronization and then a check to establish the termination condition. The termination protocol begins when a scheduler thread sends a termination request to the *manager thread*. To process a termination request, the manager thread stops each scheduler thread and then checks that all transactions are disabled. If all of the transactions are disabled, the system terminates. Otherwise, the manager thread restarts the scheduler threads. A scheduler thread may choose to request termination at any time and different heuristics may be used to determine when a scheduler thread makes the termination request. Deferring the request has the advantage of avoiding the termination protocol overhead. For example, a scheduler thread might wait until it makes a complete pass through its list of transactions without executing one before making the request.

Such a centralized termination protocol is rather disruptive as it must stop the system to check for termination. Thus, one goal may be to let a thread idle itself and be woken up by active neighbors. Similarly, the scheduler threads can check for the termination condition in parallel and wake up all of their neighbors if a transaction is enabled. Finally, the manager thread may be done away with entirely and the protocol rewritten as a distributed protocol, as we discuss below.

6.4 Scheduler Implementations

We now describe two specific scheduler designs that we chose for implementation and evaluation. Each scheduler takes a different approach to the design dimensions described previously, resulting in distinct consequences for systems that use it.

Concurrent scheduler with a global instance work queue. The first scheduler that we implemented was a concurrent scheduler with a global instance work queue. The work queue is initialized to contain all of the instances in the system. The scheduler threads take an instance from the work queue, select and execute all transactions in the instance, and insert any instances that might have changed state back into the work queue. Fairness is achieved by processing the queue in FIFO order and safety is achieved through the aforementioned locking scheme. The termination protocol for this scheduler involves counting the number of instances that are currently in the queue and the number of instances currently being processed by the scheduler threads. When this count drops to zero, there are no enabled instances in the system and the system may terminate responsibly.

One potential weakness of this scheduler is the synchronization required to coordinate access to the work queue, which adds a communication overhead and may create contention if the scheduler threads are lightly loaded, i.e., they frequently return to the queue looking for work. Thus, the scalability of this scheduler is a concern.

Concurrent partitioned round-robin scheduler with asynchronous locking and distributed termination. One of the goals when designing this scheduler was to avoid the blocking behavior and overhead of locking observed during our evaluations of the concurrent scheduler with a global instance work queue. The pthreads reader/writer locks used to protect each component instance were replaced by asynchronous reader/writer locks implemented in user-space. The asynchronous locks consist of a spin lock, variables indicating the status of the lock, and a queue of requests. To acquire a lock for a component, a scheduler thread first acquires the spin lock and then checks if the lock can be acquired immediately, which it can if either the lock has no owner or the thread is a reader (i.e., it is requesting a read lock); the lock has already been locked by another reader; and there are no writers in the queue. Otherwise, the request is placed on the queue. A scheduler thread failing to acquire the lock can either idle or attempt to execute a different transaction. When a lock is unlocked, the thread at the front of the queue is notified so it may resume processing the transaction that it was attempting to execute. The protocol avoids reader and writer starvation and ensures fairness since the queue is processed in FIFO order. This scheduler is *work conserving* since a scheduler thread continues to execute transactions instead of blocking. However, scheduler threads do not steal work from other threads.

The aforementioned centralized termination protocol has the potential to be disruptive. Thus, the motivation for a distributed termination protocol is to keep the scheduler threads as busy as possible meaning that the termination protocol is started infrequently and the termination protocol itself aborts as early as possible if the termination condition has not been reached. For this scheduler, the threads are arranged in a ring and communicate using asynchronous message queues. Messages are stamped with the id of the originating thread. The protocol forwards messages around the ring so a thread receiving a message from itself knows that all of the threads have processed that message.

Like the centralized version, the ring-based distributed termination protocol consists of a synchronization phase and checking phase. Scheduler threads begin in the RUN state where they are actively cycling through their lists of transactions. When a scheduler thread determines that termination may be possible, it sends a message to its neighbor indicating that it is entering the SYNC state. If the neighbor thread is in the RUN state, it resolves to send a reset message the next time it executes a transaction which means that the termination condition has not been established. Otherwise, the neighbor thread itself is already in the SYNC state and forwards the message. Reset messages are unconditionally forwarded around the ring and cause all threads to enter the RUN state. Synchronization messages are only forwarded if the thread receiving the message is in the SYNC state. Thus, if a thread receives its own synchronization message, all other threads in the scheduler are in the SYNC state. Multiple threads may receive their own synchronization message at the same time.

The goal of the synchronization phase is to establish a common point of reference for determining if any transaction is enabled. When a thread receives its own synchronization message, it sends a message to its neighbor that it is entering the CHECK state. This message is forwarded around the ring causing all of the threads to enter the CHECK state. The CHECK state is like the RUN state in that any executed transaction causes a reset message to circulate around the ring. A thread that cycles through its list of transactions and finds no enabled transactions sends a wait message to its neighbor and enters the WAIT state. If the neighbor is in the WAIT state, it forwards the message. If a thread receives its own wait message, then all of the threads are in the WAIT state and the termination condition has been established. Upon receiving its own wait message, a thread sends a termination message causing all threads to terminate.

6.5 Scheduler Evaluation

The utility of the reactive component model rests on the ability to execute reactive programs effectively. The challenge, then, is to design and implement effective schedulers subject to the constraints and limitations imposed by the model. The exercise of developing and evaluating schedulers may suggest possible improvements to the model or provide evidence that core features of the model resist efficient implementation. The definition of an effective scheduler will vary by platform and problem domain. For example, embedded systems may prefer a single-threaded scheduler with minimal memory requirements and power-awareness. Our focus in this work is to make progress on general-purpose concurrent schedulers. We propose the following metrics for the evaluation of a general-purpose scheduler:

Throughput: the number of transactions executed per second. Given the same system, a scheduler with higher throughput is preferable to a system with lower throughput as it is accomplishing more work per unit of time.

Latency: the amount of time between an transaction becoming enabled and being executed. The goal in measuring the latency between a transaction becoming enabled and its execution is to quantify the responsiveness of the scheduler. An interactive application may prefer a scheduler that executes enabled transactions promptly to aid in providing a good user experience or other benefit.

Utilization: the fraction of the CPUs used. Utilization is a measure of how efficiently the scheduler is using the CPUs and can be used to quantify an improvement in throughput or latency. For example, a 10% increase in throughput accompanied by a 10% increase in utilization may be acceptable while a 10% increase in throughput with a 100% increase in utilization may not be acceptable.

Both throughput and latency may be considered for individual transactions, or may be aggregated over all transactions in the system.

Evaluation approach. We used two variants of the clock system from Chapter 3 to evaluate the two scheduler implementations. The first variant is the fully-factored clock system augmented with counters that cause termination[10]. In this system, there are three components of interest: the Client, the Server, and the Counter. This system has three transactions:

- Request - The client requests the time from the server. This involves the Client and the Server.
- Response - The server responds with the time. This involves the Client, the Server, and the Counter.
- Tick - The counter increments the current time. This involves the Counter.

This system will be denoted as the *AsyncClock* system.

Figure 6.2 shows a *race graph* for the AsyncClock system. Each node in the graph is a transaction $u \in T$. Let $acc(u) = pre(u) \cup imm(u) \cup mut(u)$. An edges exists between

[10]https://github.com/jrw972/rcgo/blob/master/samples/clock3.rc

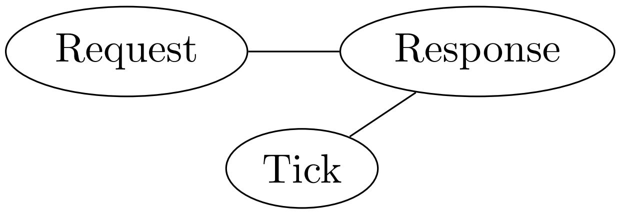

Figure 6.2: Race graph for the AsyncClock system. Each node is a transaction. Nodes sharing an edge cannot be executed concurrently due to potentially mutated shared state.

two nodes $u, v \in T$ if $race(acc(u), acc(v))$. Nodes sharing an edge cannot be executed concurrently due to potentially mutated shared state. Conversely, nodes that are not linked by an edge can be executed concurrently. For the purpose of evaluating schedulers, the clock system has the important characteristic that Request and Tick can be executed concurrently while Response must be executed independently. Another useful feature of this system is that each component has exactly one transaction, and thus, a work queue of instances may also be viewed as a work queue of transactions.

The second variant is a simplified version of the clock system consisting of a Tick transaction that increments the counter and a Request transaction that uses a getter to sample the counter[11]. The counter creates a race between the two transactions so that they cannot be executed concurrently. This system will be denoted as the *SyncClock* system.

For comparison, we implemented multi-threaded versions of the AsyncClock and SyncClock systems using the POSIX threads library (pthreads). The AsyncClock implementation attempts to preserve the spirit of the AsyncClock system by being as asynchronous as possible. This implementation uses three threads corresponding to the Request, Response, and Tick transactions. The Request and Response threads share a flag variable that is protected by a mutex and condition variable to implement the asynchronous request/response protocol. The Response and Tick threads share a counter variable that is protected by a mutex. The SyncClock implementation attempts to preserve the spirit of the SyncClock system. In this design, there is one thread sampling the counter while another thread is incrementing the counter. The two threads synchronize access to the counter through a single mutex. For convenience, experiments involving a pthreads implementation will be labeled *Thread*, experiments involving the concurrent scheduler with a global instance work queue will be labeled

[11]https://github.com/jrw972/rcgo/blob/master/samples/clock4.rc

Instance, and experiments involving the concurrent partitioned round-robin scheduler with asynchronous locking and distributed termination will be labeled *Partitioned*.

The AsyncClock and SyncClock systems were executed 1,000 times to profile the performance of each scheduler. The programs are instrumented with profiling code that records the total execution time and timestamps for each transaction. The timestamps are stored in memory and written after the scheduler terminates to avoid disruptions in timing due to output. The Request and Tick transactions were limited to 10,000 executions. Thus, each SyncClock run generates 20,000 data points. In the AsyncClock system, the relationship between Request and Response causes Response to be executed 10,000 times as well. Thus, each AsyncClock run generates 30,000 data points. The throughput for a single run is determined by dividing the total number of transactions (20,000 or 30,000) by the total time used for execution. The utilization was determined using the `time` utility and the number of context switches was determined using the `getrusage` system call before and after the execution of the system. The latency for each transaction is determined by computing the difference between the transaction start time and the end time of the enabling transaction. For the SyncClock system, this is $Request_{t+1} - Request_t$ and $Tick_{t+1} - Tick_t$. For the AsyncClock system, this is $Request_{t+1} - Response_t$, $Response_{t+1} - Request_t$, and $Tick_{t+1} - Tick_t$. For the SyncClock system, $1,000 \times 20,000 = 20,000,000$ latency points were collected. For the AsyncClock system, $1,000 \times 30,000 = 30,000,000$ latency points were collected.

Placing all transactions in a run on a timeline according to their start times, we define the *entanglement* of a run to be the number of adjacent transactions on the timeline that were executed in different threads. The entanglement is used to measure the granularity of the concurrency between the scheduler threads. Given that each run executes the same number of transactions, a high entanglement indicates “fine” concurrency, i.e., threads are executing transactions in parallel or execution is rapidly alternating between the threads, while a low entanglement indicates “coarse” concurrency. Entanglement may be forced by the scheduler, or may occur opportunistically, or not at all. The start time for threads influences entanglement as an operating system may execute a thread to completion before starting another thread. The garbage collection transactions are not included in the entanglement calculation to facilitate comparison with the pthread implementations.

Machine Model:	Lenovo G570 Laptop
Operating System:	Ubuntu 14.04
Processor:	Intel Pentium B960 2.20GHz
Architecture:	64-bit
Cores:	2
Memory:	4GB
Kernel:	3.13.0-77-generic #121-Ubuntu SMP
Compiler:	g++ 4.8.4
C library:	glibc 2.19
POSIX threads:	glibc 2.19
C++ library:	glibc++ 3.4.19

Table 6.1: Experimental environment used for scheduler testing

Table 6.1 describes the environment used to perform the scheduler experiments[12]. The timestamp resolution reported by the operating system was 1 ns. The Instance and Partitioned schedulers were configured to use two threads. The Thread applications for the AsyncClock and SyncClock systems were designed to use three and two threads, respectively. The raw data for the experiments is available[13].

[12]The version of the code used for this evaluation bears the tag "clock_experiment2" and can be found at https://github.com/jrw972/rcgo/releases/tag/clock_experiment2.

[13]https://drive.google.com/open?id=0BwSc4YwTjv7TOHdVMENkU000WUU

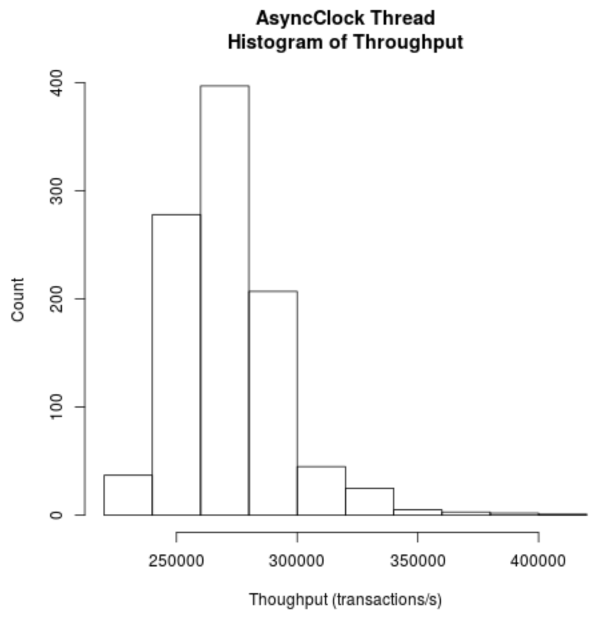

Figure 6.3: AsyncClock Thread Histogram of Throughput

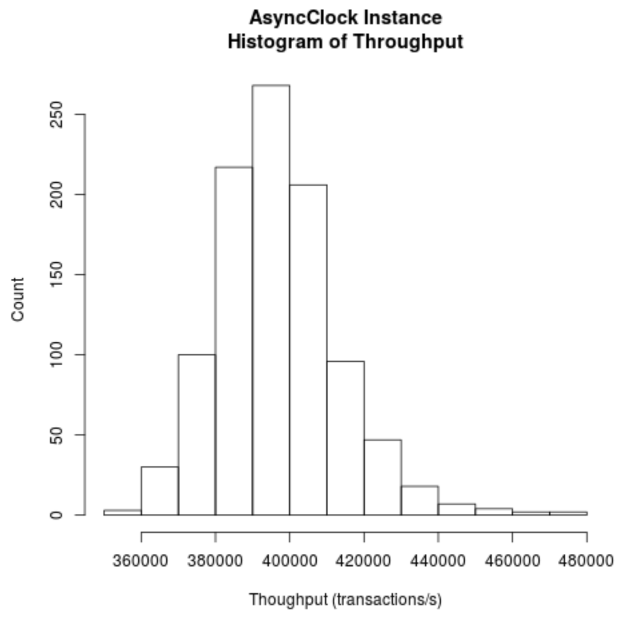

Figure 6.4: AsyncClock Instance Histogram of Throughput

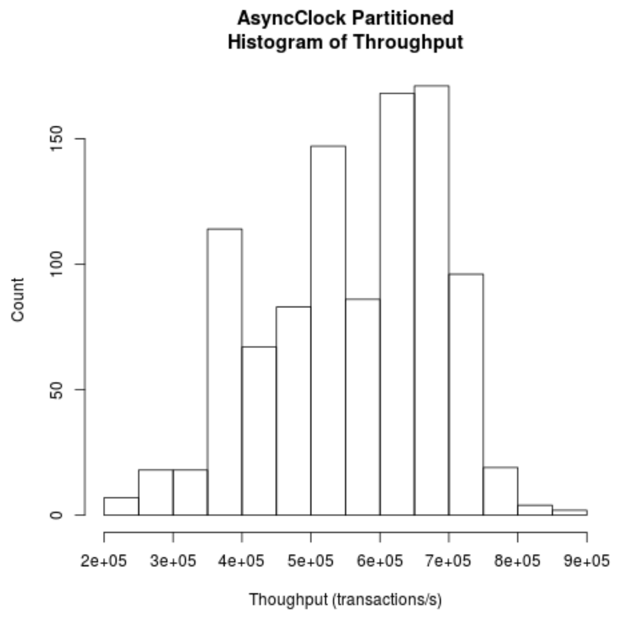

Figure 6.5: AsyncClock Partitioned Histogram of Throughput

AsyncClock results. Figures 6.3, 6.4, and 6.5 show histograms of the throughput for the Thread, Instance, and Partitioned experiments, respectively. The throughput for the Thread and Instance experiments appear to have “regular” distributions while the throughput for the Partitioned experiment is “irregular.” Transactions are randomly and statically allocated to execution threads in the Partitioned experiment. In the AsyncClock system, there are seven transactions where three correspond to the Request, Response, and Tick transactions, and the other four correspond to garbage collection transactions for the Client, Server, Counter, and System components. For convenience, garbage collection transactions will be named by the corresponding component. Thus, there are $2^7 = 128$ mappings of transactions to execution threads. Given the symmetry of threads, this results in 64 possible arrangements which are shown in Appendix A in Table A.1. Some of these arrangements will place all transactions in one thread, leading to improved or degraded performance, e.g., ‘A’. Some arrangements will place the Request and Tick transactions in different threads, allowing the potential concurrency among these transactions to be exploited, e.g., ‘B’. Some arrangements will place the Tick and Counter transactions on different threads, creating lock contention, e.g., ‘C’. Thus, the throughput for the Partitioned experiment is a mixture of distributions corresponding to the different partitioning schemes.

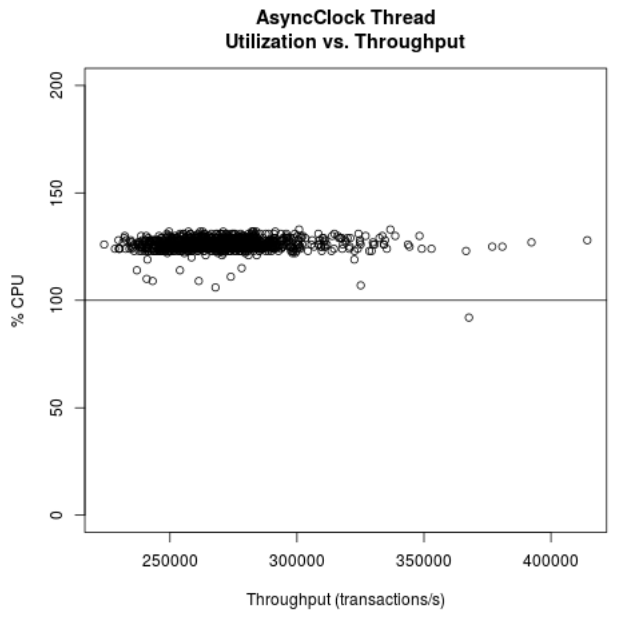

Figure 6.6: AsyncClock Thread Utilization vs. Throughput

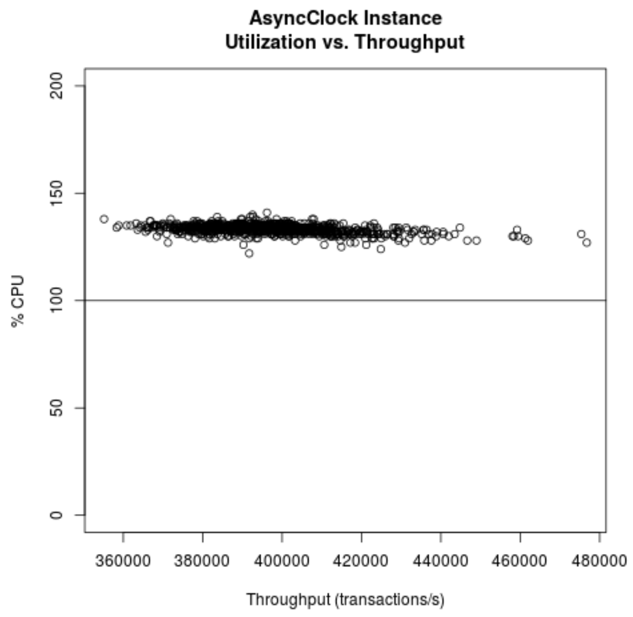

Figure 6.7: AsyncClock Instance Utilization vs. Throughput

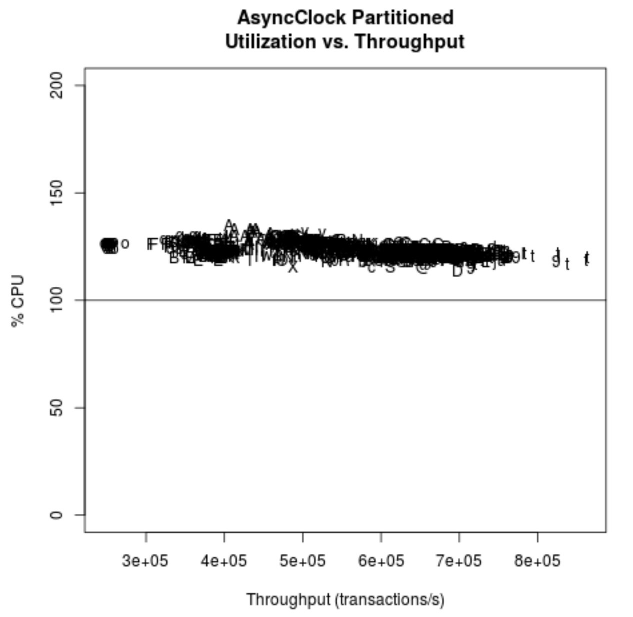

Figure 6.8: AsyncClock Partitioned Utilization vs. Throughput

Figures 6.6, 6.7, and 6.8 show plots of utilization versus throughput for the Thread, Instance, and Partitioned experiments, respectively. Most of the runs achieve a utilization greater than 100%. For the Instance and Partitioned experiments, there appears to be slight trend where utilization decreases with increased throughput. This is most likely the result of locking behavior where runs that (randomly) experience better locking behavior simultaneously achieve higher throughput and lower utilization due to less locking overhead.

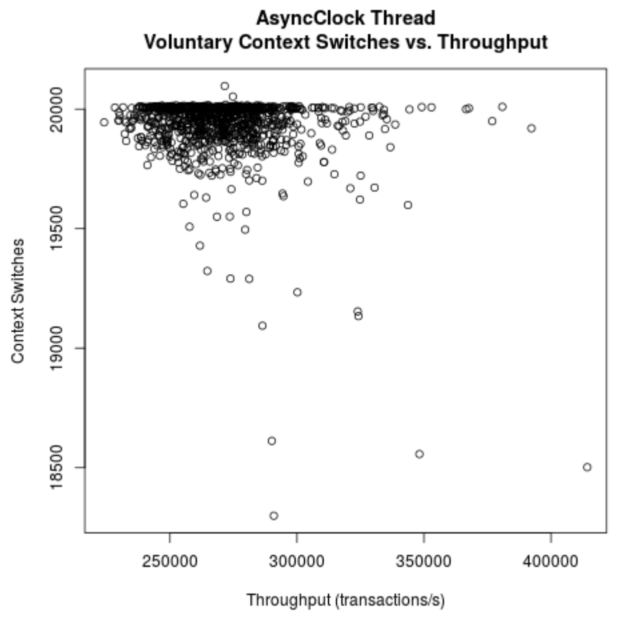

Figure 6.9: AsyncClock Thread Voluntary Context Switches vs. Throughput

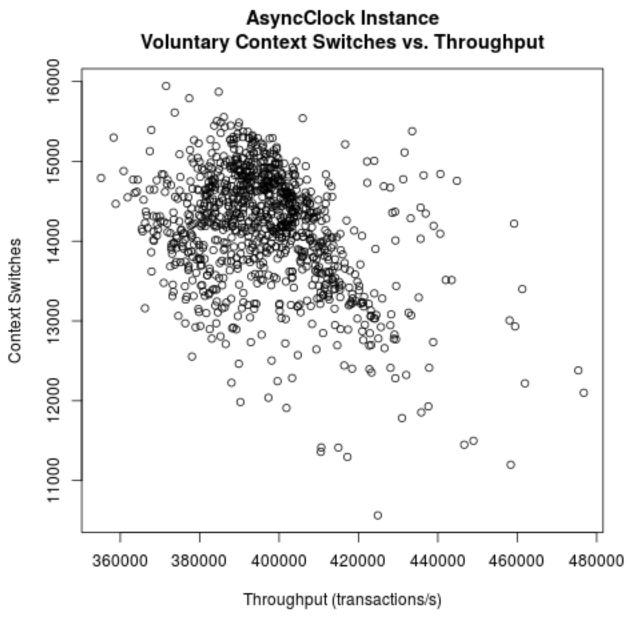

Figure 6.10: AsyncClock Instance Voluntary Context Switches vs. Throughput

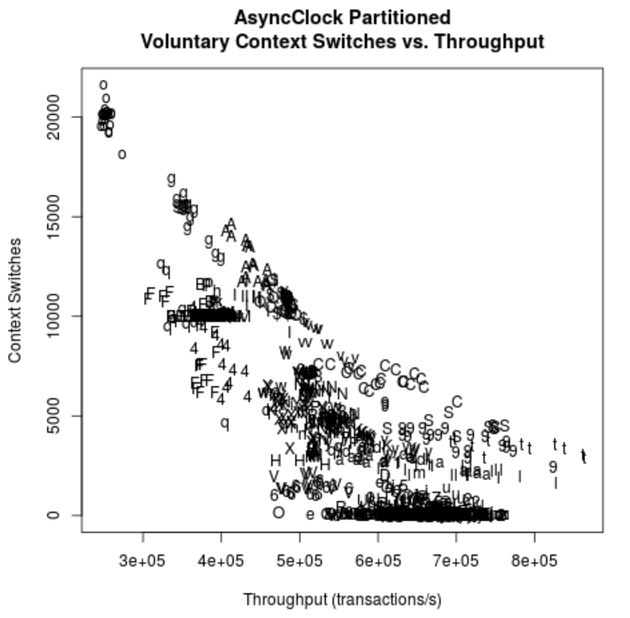

Figure 6.11: AsyncClock Partitioned Voluntary Context Switches vs. Throughput

Figures 6.9, 6.10, and 6.11 show plots of voluntary context switches versus throughput for the Thread, Instance, and Partitioned experiments, respectively. Voluntary context switches include the situation where a thread blocks while waiting for a lock and is swapped out. The Thread experiment appears to have a consistently high number of context switches. This is expected since there are three threads competing for two processors. The Instance experiment has fewer context switches and perhaps shows a slight trend where throughput increases with fewer context switches. As previously stated, this is most likely the result of locking behavior. The Partitioned experiment shows a strong trend of throughput increasing with fewer context switches. Furthermore, the partitions appear to cluster together. For example, all of the runs for partition ‘o’ of Table A.1 appear in the upper left of Figure 6.11, while the runs for partition ‘t’ appear in the lower right. In partition ‘o’, thread 0 contains the System, Counter, Tick, Request, and Response transactions while thread 1 contains the Server and Client transactions. This mapping serializes the execution (the Request, Response, and Tick transactions are on one thread) and is subject to contention arising from the Server and Client transactions competing with the Request and Response transactions. In partition ‘t’, thread 0 contains the System, Counter and Tick transactions while thread 1 contains the Server, Client, Response, and Request transactions. The only contention in this mapping is between the Response transaction in thread 1 and the Counter and Tick transactions in thread 0.

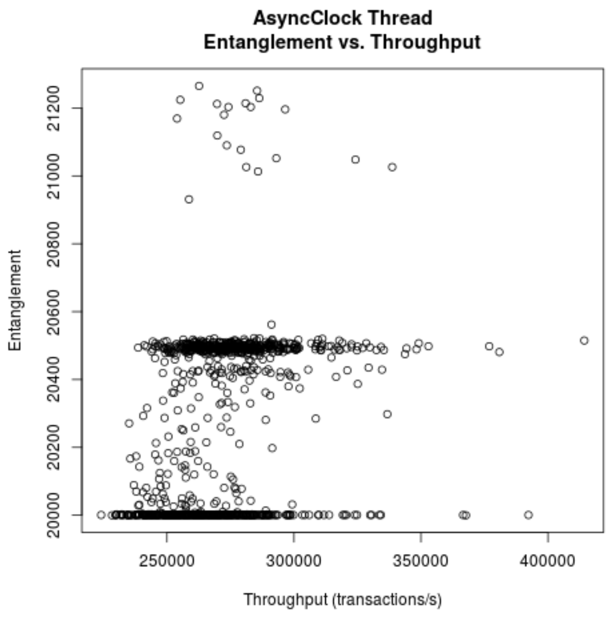

Figure 6.12: AsyncClock Thread Entanglement vs. Throughput

Figure 6.12 shows a plot of entanglement versus throughput for the Thread experiment. The Thread experiment has a minimum entanglement of 20,000, which is caused by the forced interleaving of the Request and Response threads, and a maximum entanglement of 30,000. Thus, the lowest band in Figure 6.12 corresponds to no entanglement with the Tick thread. In these cases, the Linux thread scheduler executes the Request/Response threads first and the Tick thread second (or vice versa). The generally low entanglement shows that the Linux scheduler prefers to serialize the execution. This is reasonable given that the Thread scheduler is subject to a performance hit caused by context switches as previously described. The upper band suggests a bound on the allowed entanglement. One explanation for this behavior is that the Linux scheduler may limit context switches, which has the effect of limiting the entanglement in the Thread scheduler. Another explanation is that the entanglement may be limited based on the threads starting at different times.

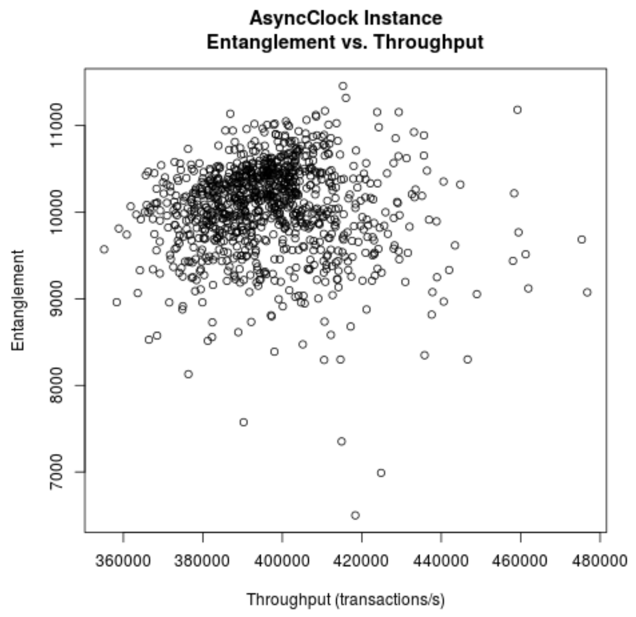

Figure 6.13: AsyncClock Instance Entanglement vs. Throughput

Figure 6.13 shows a plot of entanglement versus throughput for the Instance experiment. The minimum entanglement is 0 and the maximum entanglement is 30,000[14]. This plot does not appear to show a strong relationship between throughput and entanglement.

[14]Let Transaction(Thread) indicate that the transaction was executed by the corresponding thread. The following pattern when repeated 5,000 times generates an entanglement of 30,000: Request(0) Tick(1) Response(0) Request(1) Tick (0) Response (1).

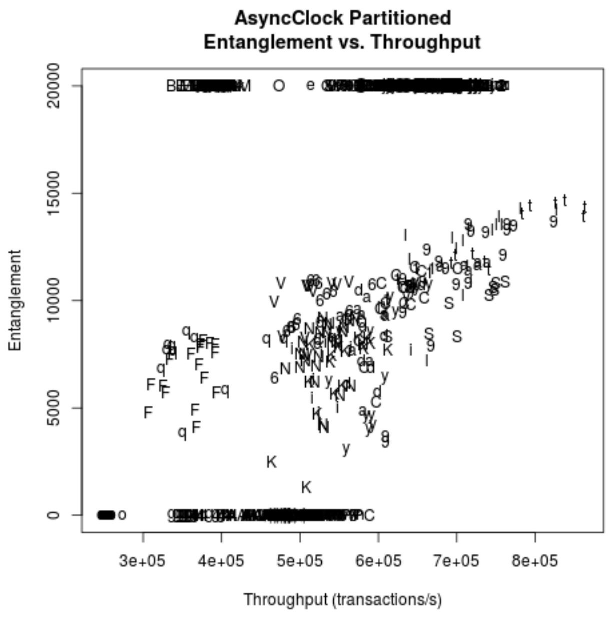

Figure 6.14: AsyncClock Partitioned Entanglement vs. Throughput

Figure 6.14 shows a plot of entanglement versus throughput for the Partitioned experiment. This plot has a band at 20,000 that contains roughly half of the samples (513/1000). These correspond to arrangements where the Request and Response transactions are mapped to different execution threads which forces an entanglement of 20,000. Half of the remaining samples have no entanglement (246/1000). These correspond to arrangements where the Request, Response, and Tick transactions are mapped to the same execution thread where no entanglement is possible. The remaining quarter of the samples (241/1000) correspond to arrangements where the Request and Response transactions are in one execution thread while the Tick transaction is in the other. These show varying degrees of entanglement with perhaps a slight trend toward increasing throughput with greater entanglement. This is expected because the concurrency between Request and Tick can be exploited in these arrangements.

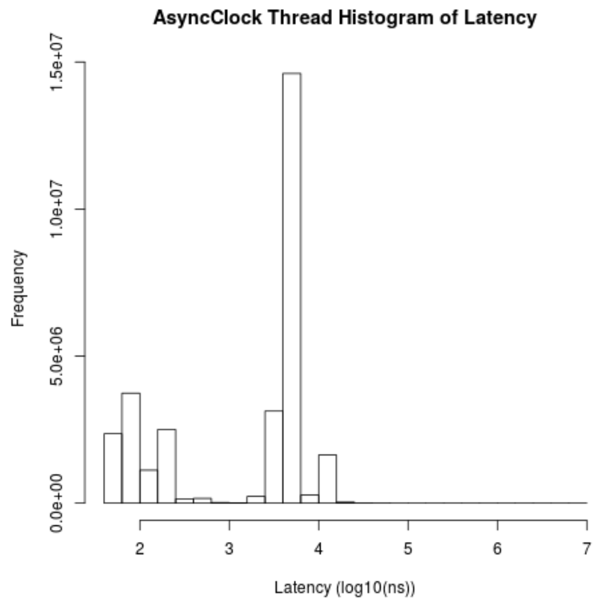

Figure 6.15: AsyncClock Thread Histogram of Latency

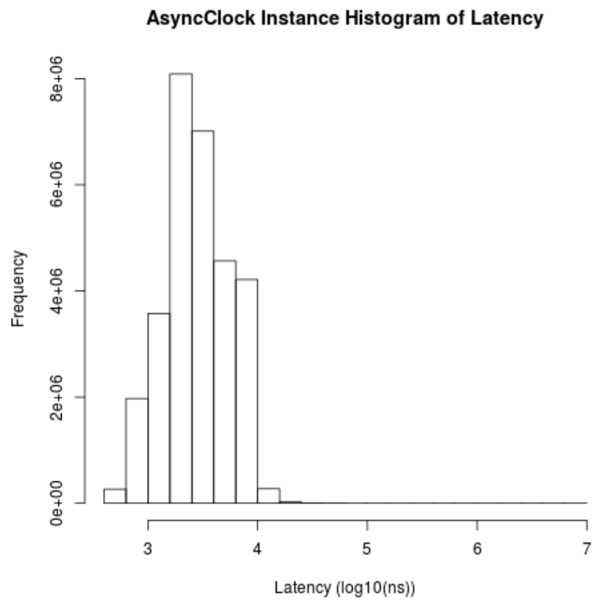

Figure 6.16: AsyncClock Instance Histogram of Latency

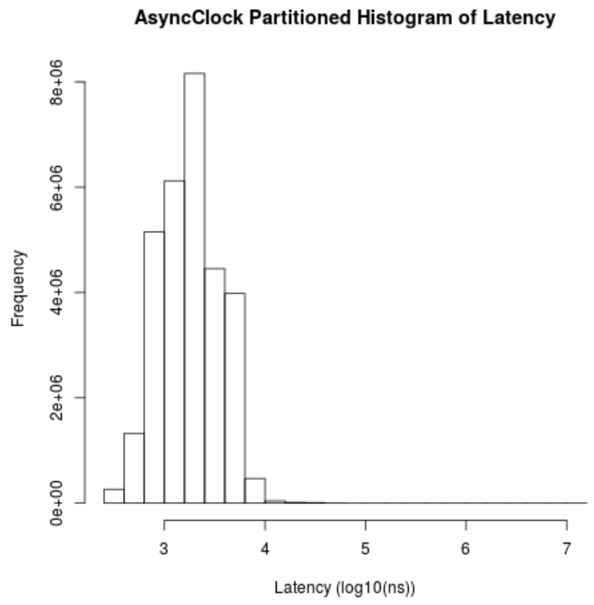

Figure 6.17: AsyncClock Partitioned Histogram of Latency

Figures 6.15, 6.16, and 6.17 show histograms of the latency for the Thread, Instance, and Partitioned experiments, respectively. All plots of latency use a logarithmic x-axis, as the latency distributions have very long tails. For the Thread experiment, the mean latency of the transactions are as follows:

Tick	0.11769us
Request	5.30299us
Response	5.63980us

Thus, the distribution on the left in Figure 6.15 corresponds to the latency of the Tick transaction while the distribution on the right corresponds to the latency of the Request and Response transactions. In the Thread experiment, Tick transactions can be executed in quick succession as they are in a tight loop. For the Instance experiment, the mean latency of the transactions are as follows:

Tick	5.22393us
Request	2.83083us
Response	2.58799us

The Instance scheduler tends to serialize the execution of transactions for the AsyncClock system which has the effect of minimizing race conditions. Thus, the Tick transaction has roughly twice the latency of the Request and Response transactions since the Tick transaction is always enabled while the Request and Response transactions are enabled by each other. For the Partitioned experiment, the mean latency of the transactions are as follows:

Tick	2.49278us
Request	2.31450us
Response	2.06478us

The work-conserving nature of the Partitioned scheduler allows it to achieve a lower average latency for all transactions when compared to the Instance scheduler.

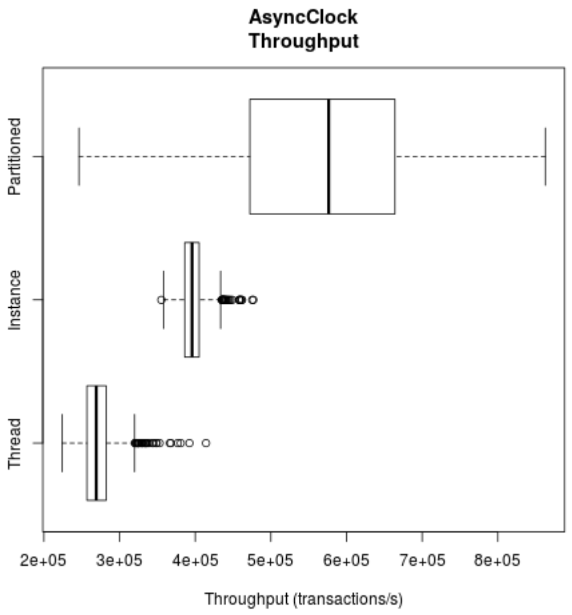

Figure 6.18: AsyncClock Throughput

Figure 6.18 shows a box plot of the throughput for the Thread, Instance, and Partitioned experiments. The quantiles of the throughput in transactions/s for each scheduler are as follows:

Scheduler	0%	25%	50%	75%	100%
Partitioned	246,370.0	472,679.2	576,411.0	664,042.5	862,717.0
Instance	355,158.0	386,067.8	395,796.0	405,083.8	476,759.0
Thread	224,107.0	256,876.0	269,193.5	282,329.0	414,293.0

The work-conserving nature of the Partitioned scheduler allows it to achieve a higher throughput than the Instance scheduler, and its ability to avoid context switches allows it to achieve a higher throughput than the Thread scheduler. However, the results also demonstrate the variability in the throughput due to the many modes of partitioning. Thus, the Partitioned scheduler could be improved by using the race graph to avoid "bad" partitions.

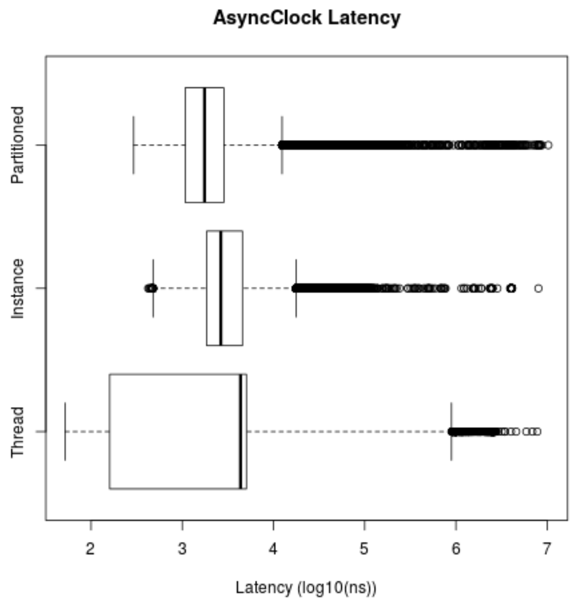

Figure 6.19: AsyncClock Latency

Figure 6.19 shows a box plot of the latency for the Thread, Instance, and Partitioned experiments. The quantiles of the latency (in ns) for each scheduler are as follows:

Scheduler	0%	25%	50%	75%	100%
Partitioned	293	1,075	1,754	2,861	10,296,400
Instance	423	1,851	2,645	4,569	8,028,100
Thread	52	160	4,352	5,051	7,791,400

The Thread scheduler achieves the lowest latency via the Tick transaction due to its work-conserving nature and optimized locking scheme.

Both the Instance and Partitioned schedulers contain transactions for performing garbage collection. These transactions do not actually perform any work since none of the transactions in the AsyncClock system allocate memory. The Instance scheduler executed an average of 33,053 garbage collection actions per run while the Partitioned scheduler executed an average of 97,078 garbage collection actions per run. The excessive number of collections performed by the Partitioned scheduler shows a potential problem between the garbage-collection-as-an-action idea and work conserving schedulers, as a work conserving scheduler can always find more work to do in garbage collection attempts. Ideally, a scheduler would be designed to not even select a garbage collection action until it has a high probability of actually collecting garbage.

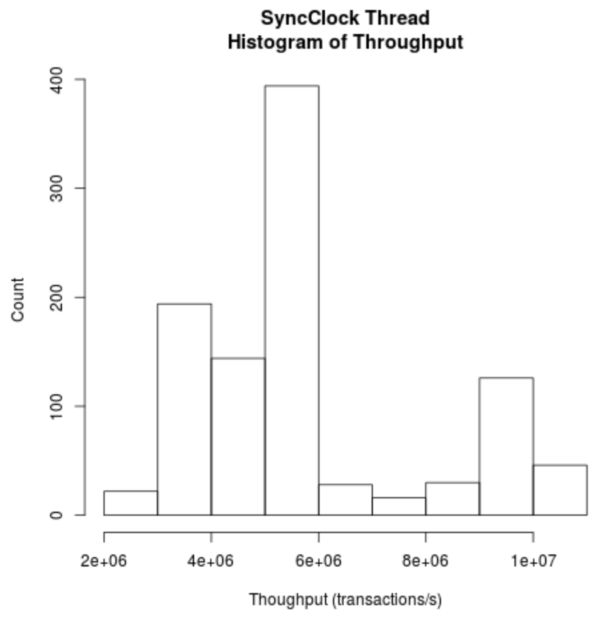

Figure 6.20: SyncClock Thread Histogram of Throughput

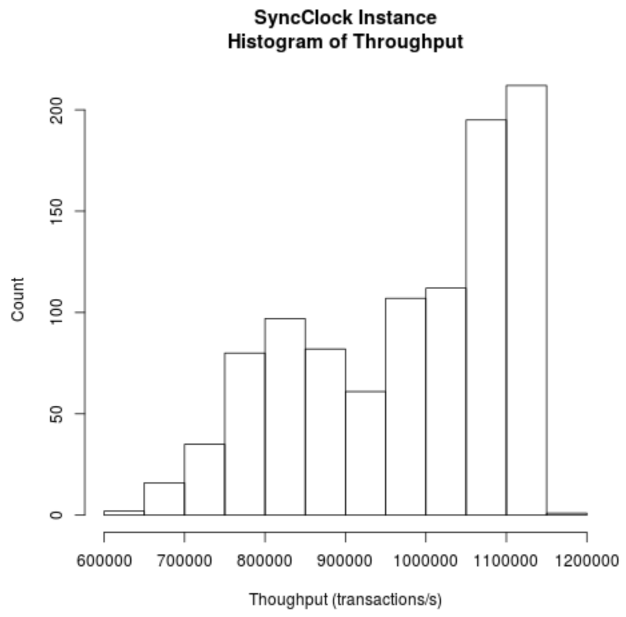

Figure 6.21: SyncClock Instance Histogram of Throughput

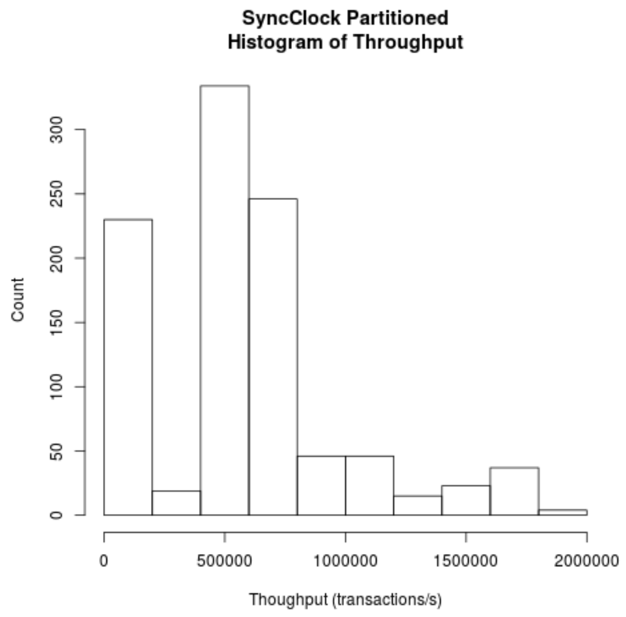

Figure 6.22: SyncClock Partitioned Histogram of Throughput

SyncClock results. Figures 6.20, 6.21, and 6.22 show histograms of the throughput for the Thread, Instance, and Partitioned experiments, respectively. The throughput curves of the Thread and Instance experiments appear to be combinations of two or three distributions. As in the AsyncClock experiments, the throughput for the Partitioned experiment is a mixture of distributions arising from different partitioning schemes. In the SyncClock system, there are two components and two transactions resulting in 8 possible partitions. The list of partitions is given in Appendix A in Table A.2.

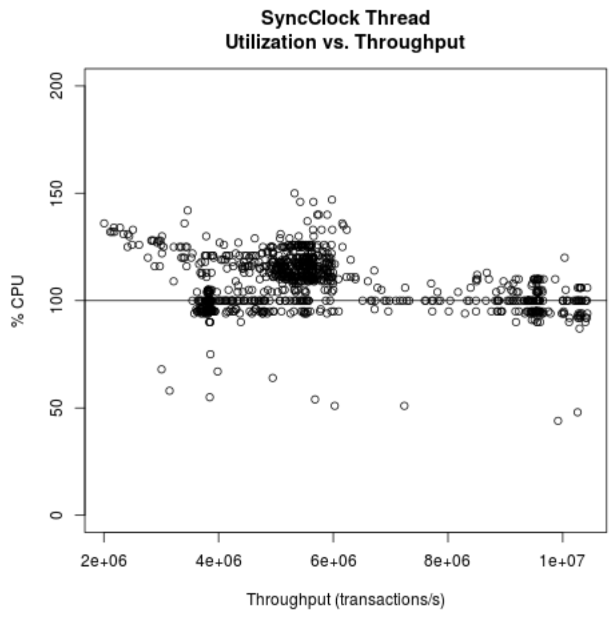

Figure 6.23: SyncClock Thread Utilization vs. Throughput

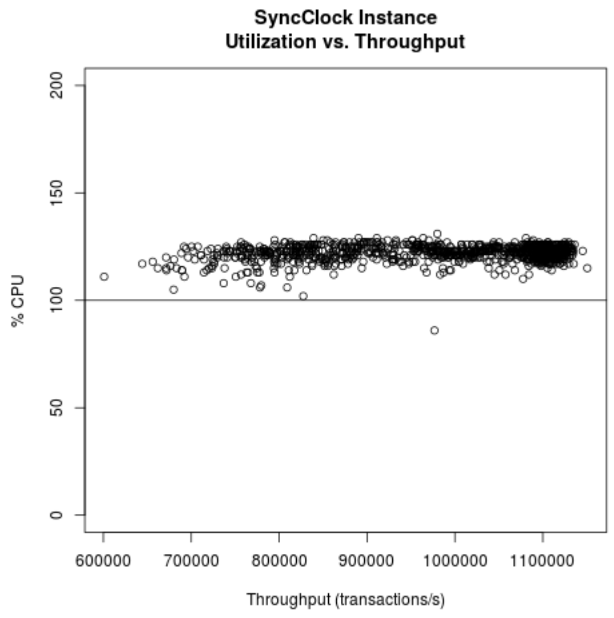

Figure 6.24: SyncClock Instance Utilization vs. Throughput

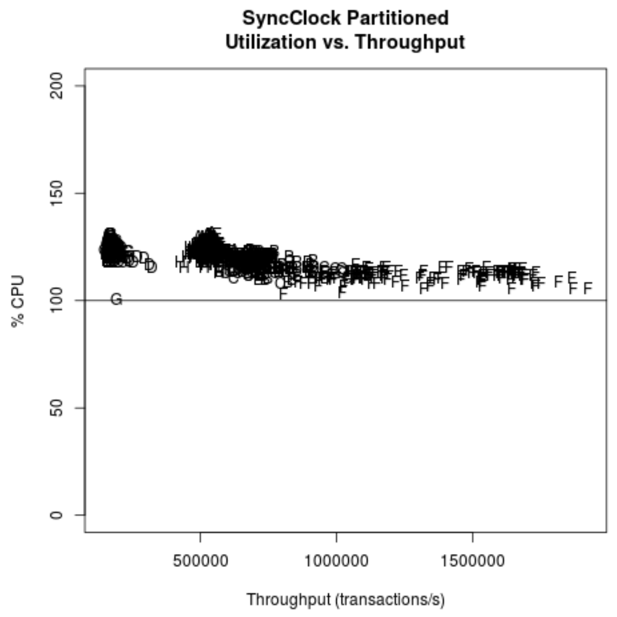

Figure 6.25: SyncClock Partitioned Utilization vs. Throughput

Figures 6.23, 6.24, and 6.25 show plots of utilization versus throughput for the Thread, Instance, and Partitioned experiments, respectively. The utilization for the Thread experiment decreases with increasing throughput. This suggests that the Linux scheduler is serializing the execution of the threads which decreases utilization while avoiding lock contention which results in increased throughput. Most of the runs for the Instance and Partitioned experiments achieve a utilization greater than 100% while a number of runs in the Thread experiment do not. The Instance experiment perhaps shows a weak trend of increased utilization with throughput while the Partitioned experiment shows a trend of decreased utilization with throughput.

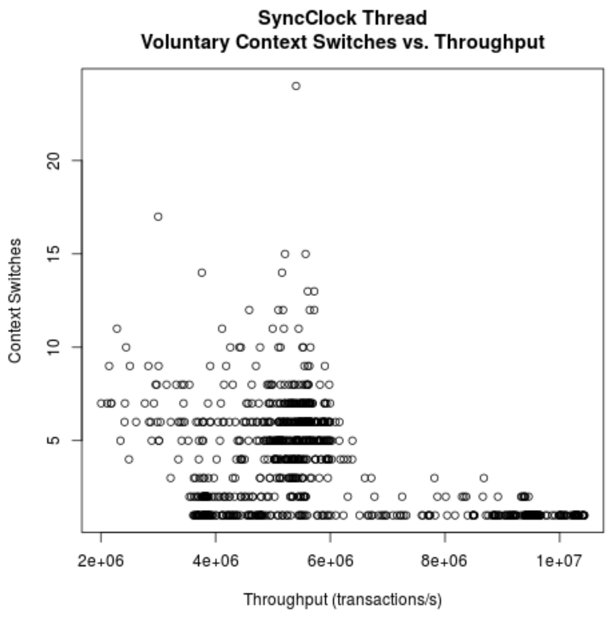

Figure 6.26: SyncClock Thread Voluntary Context Switches vs. Throughput

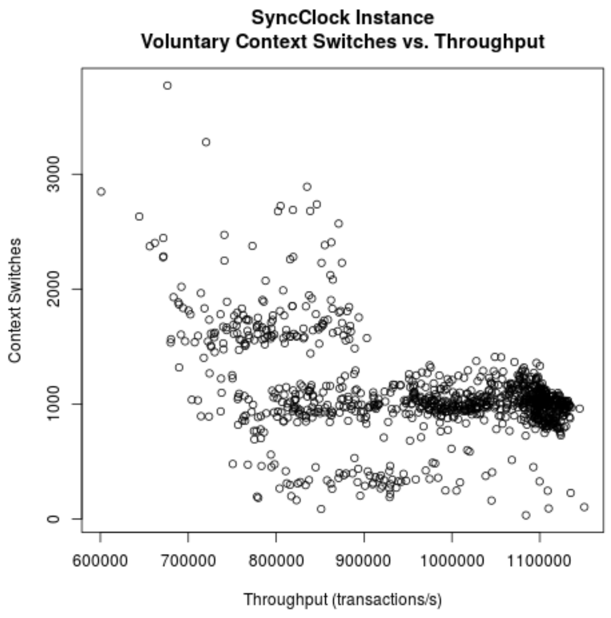

Figure 6.27: SyncClock Instance Voluntary Context Switches vs. Throughput

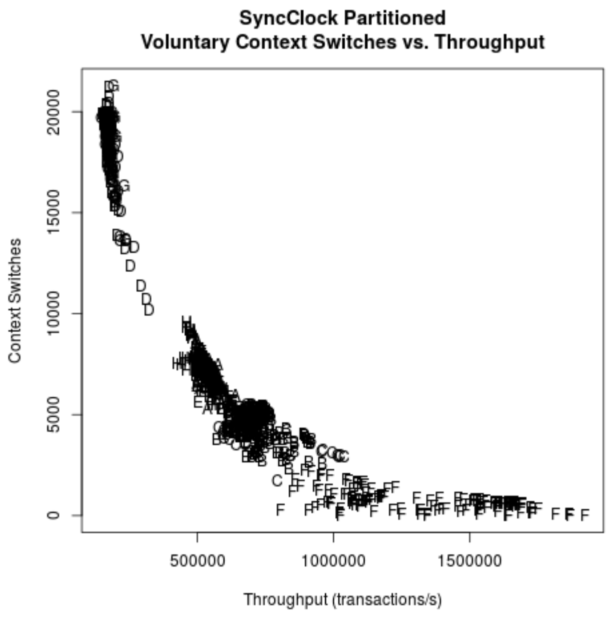

Figure 6.28: SyncClock Partitioned Voluntary Context Switches vs. Throughput

Figures 6.26, 6.27, and 6.28 show plots of voluntary context switches versus throughput for the Thread, Instance, and Partitioned experiments, respectively. All show a trend where throughput increases with decreasing context switches. Both the Thread and Instance experiments have a threshold where high throughput appears to require minimizing the number of context switches. The plot for the Partitioned experiment clearly shows an inverse relationship between throughput and context switches.

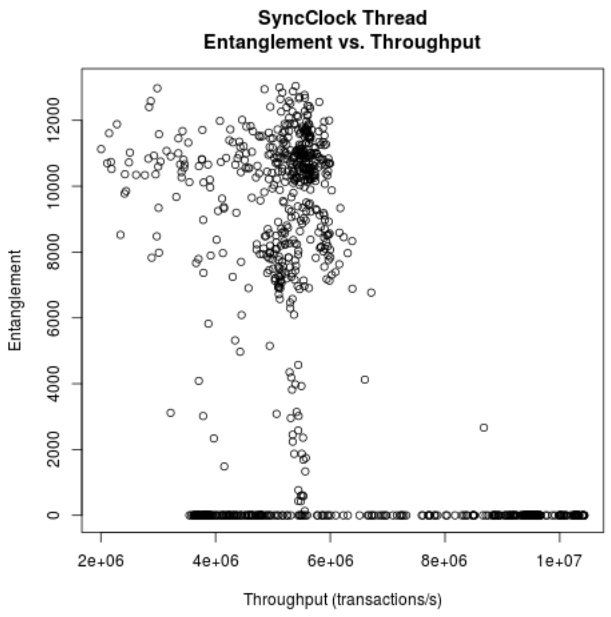

Figure 6.29: SyncClock Thread Entanglement vs. Throughput

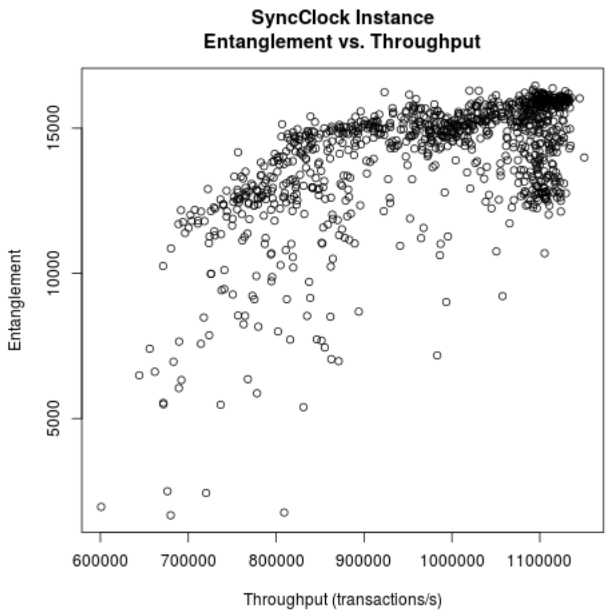

Figure 6.30: SyncClock Instance Entanglement vs. Throughput

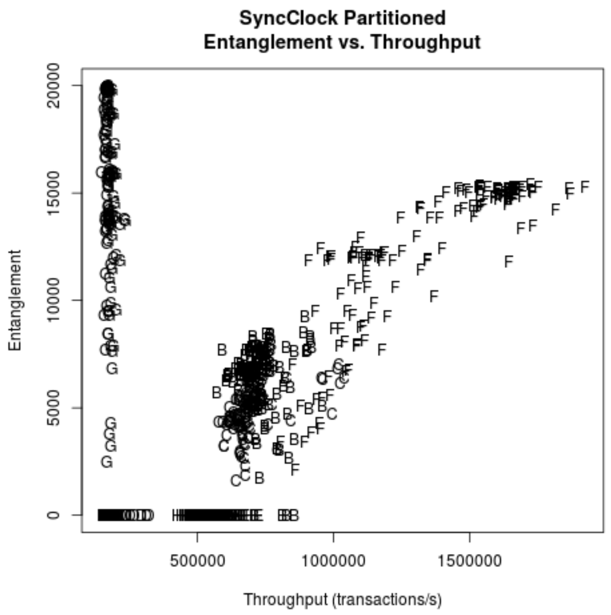

Figure 6.31: SyncClock Partitioned Entanglement vs. Throughput

Figures 6.29, 6.30, and 6.31 show plots of entanglement versus throughput for the Thread, Instance, and Partitioned experiments, respectively. The maximum entanglement for the SyncClock system is 20,000. The plot of entanglement for the Thread experiment resembles the plots of utilization and context switches. The samples appear to be divided between concurrent (high entanglement, high context switch) and serial executions (low entanglement, low context switch). For this system, it seems that it is more efficient to serialize the execution of the threads than to execute the threads concurrently and suffer context switches.

The plot of entanglement for the Instance experiment shows a different trend where throughput increases with entanglement. Thus, the Instance experiment achieves high throughput when execution alternates between the threads. The entanglement for the Partitioned experiment appears to be a combination of three distributions. The vertical line on the left consists of samples from the 'G' partition which has maximal inter-thread conflicts and minimal opportunities for concurrent execution. The horizontal line on the bottom (no entanglement) consists of samples from the 'A', 'D', 'E', and 'H' partitions which map the Tick and Request transactions to the same thread. The execution is serialized with varying degrees of interference from the garbage collection transactions. The other samples contain the 'B', ‘C’, and ‘F’ partitions which appear to show increasing throughput with entanglement. The ‘F’ partition has minimal inter-thread conflicts with maximal opportunities for concurrent execution.

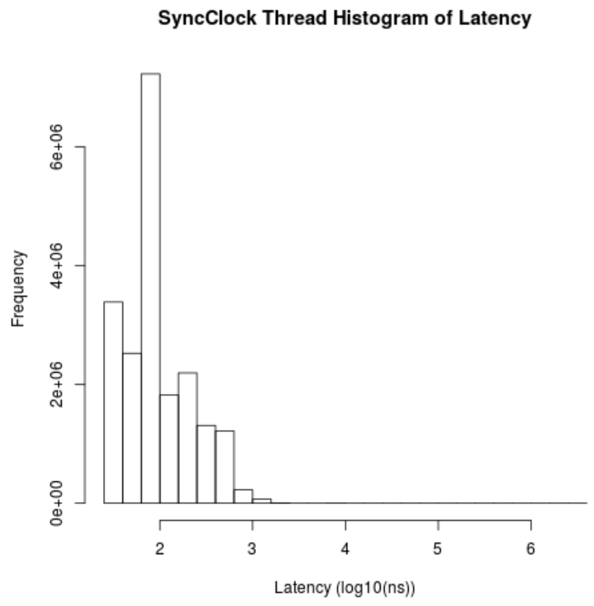

Figure 6.32: SyncClock Thread Histogram of Latency

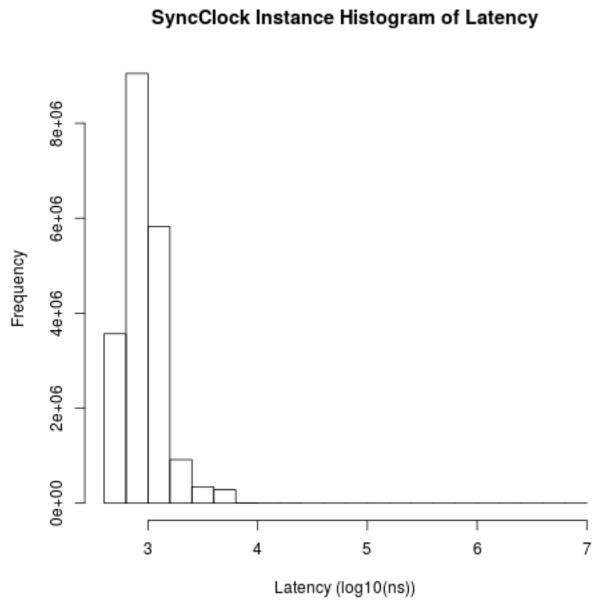

Figure 6.33: SyncClock Instance Histogram of Latency

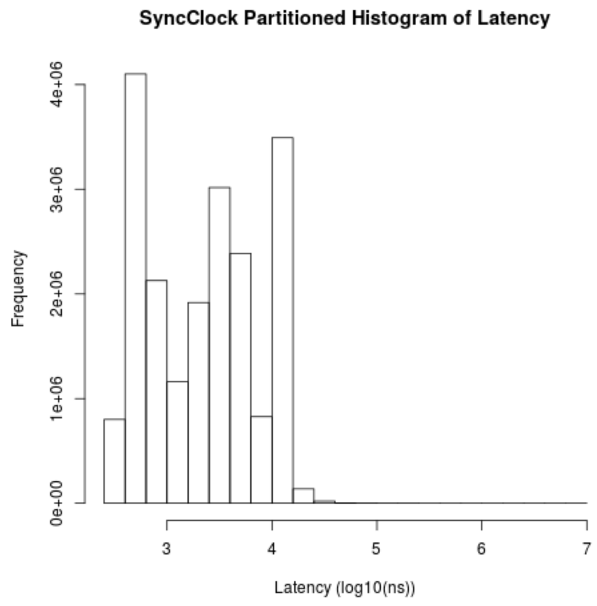

Figure 6.34: SyncClock Partitioned Histogram of Latency

Figures 6.32, 6.33, and 6.34 show histograms of the latency for the Thread, Instance, and Partitioned experiments, respectively. All plots of latency use a logarithmic x-axis, as the latency distributions have very long tails. For the Thread experiment, the mean latency of the transactions are as follows:

Request	72.952ns
Tick	207.145ns

The Request transaction does nothing more than acquire and release a lock which may explain its reduced latency. For the Instance experiment, the mean latency of the transactions are as follows:

Request	1,085.32ns
Tick	1,070.57ns

For the Partitioned experiment, the mean latency of the transactions are as follows:

Request	4,117.71ns
Tick	4,069.06ns

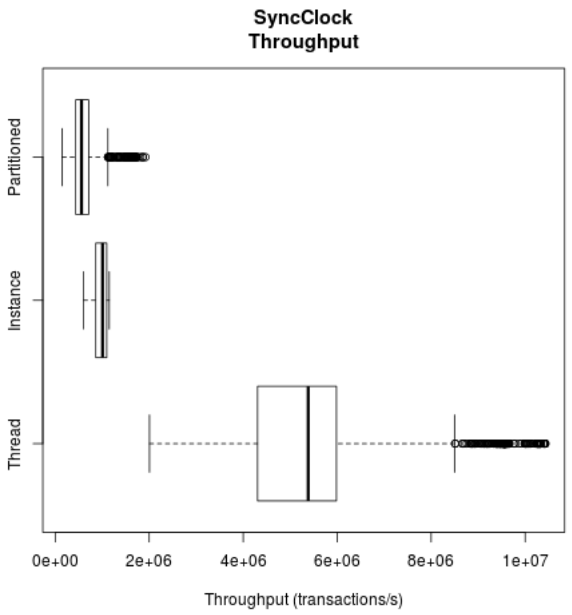

Figure 6.35: SyncClock Throughput

Figure 6.35 shows a box plot of the throughput for the Thread, Instance, and Partitioned experiments. The quantiles of the throughput in transactions/s for each scheduler are as follows:

Scheduler	0%	25%	50%	75%	100%
Partitioned	146,788.0	439,051.0	558,954.5	711,073.8	1,919,680.0
Instance	600,871.0	862,315.8	1,007,265.0	1,095,980.0	1,150,600.0
Thread	2,005,940.0	4,308,048.0	5,382,440.0	5,983,532.0	10,428,200.0

The Thread scheduler is clearly the best in terms of throughput. The best partitions of the Partitioned scheduler almost achieve the worst performance of the Thread scheduler. In aggregate, however, the Instance scheduler appears to be better than the Partitioned scheduler.

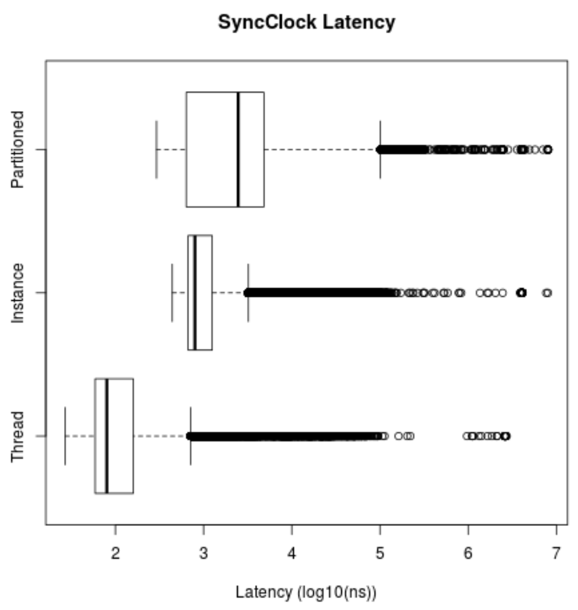

Figure 6.36: SyncClock Latency

Figure 6.36 shows a box plot of the latency for the Thread, Instance, and Partitioned experiments. The quantiles of the latency (in ns) for each scheduler are as follows:

Scheduler	0%	25%	50%	75%	100%
Partitioned	293	636	2,469	4,835	8,100,990
Instance	442	668	798	1,254	8,029,550
Thread	27	59	80	160	2,736,960

The Thread scheduler has the best latency by an order of magnitude. This is unsurprising given the efficiency of the Thread implementation.

6.6 Summary

Perhaps the most interesting part of the implementation of the rc_{go} run-time system is the scheduler. Fairness, safety, and responsibility are identified as the three essential requirements for any scheduler and we illustrate how these may be accomplished in a variety of designs. Two multi-threaded schedulers were implemented. The Instance scheduler is based on a shared work queue of instances while the Partitioned scheduler is based on partitioning transactions among different scheduler threads. We used throughput, latency, and utilization as metrics for evaluating schedulers for reactive components and collected data for the Instance and Partitioned schedulers by executing the AsyncClock and SyncClock systems. The reactive component schedulers were then compared to custom implementations of the same systems using the pthreads library.

The AsyncClock and SyncClock experiments demonstrate both the viability of reactive components as an event system and that more work is necessary to improve the design and implementation of the run-time system. The motivation for events was to facilitate (logical) concurrency while avoiding the overhead of context switching associated with assigning each task to a thread. The performance of the Partitioned scheduler over the Thread implementation for the AsyncClock system demonstrates this idea. As was previously mentioned, events can be combined with multi-threading but care must be taken to ensure proper synchronization. In the reactive component model, the burden of correct synchronization is placed on the scheduler instead of the developer.

The AsyncClock system is illustrative in that it shows the advantage of events over threads but is nevertheless unrealistic as it intentionally overloads the system. The SyncClock system represents a scenario in which there are adequate resources for each thread. In this situation, the Instance and Partitioned schedulers perform an order of magnitude worse than the Thread implementation. This prompts the question: can reactive components be as efficient as threads? While we cannot answer this question definitively, the act of converting a reactive component program to a threaded program does provide evidence that it may be possible to make reactive components as efficient as threads. The procedure for converting a reactive component program to a threaded program involves 1) creating a reader/writer lock for each component instance, 2) creating a thread for each transaction, 3) creating a condition variable and mutex for each precondition, 4) creating a critical section for each transaction, and 5) signaling affected transactions after the critical section. It seems feasible that this procedure can be automated. Thus, a system of reactive components can be converted to threads or scheduled as events depending on the resources available and the nature of the transactions in the system, but doing so is left for future work.

Chapter 7

Conclusions and Future Work

A reactive system is characterized by "ongoing interactions with its environment" [66]. Asynchronous concurrency is a feature of reactive systems that makes them inherently difficult to develop. Reactive systems are already used in various forms of critical infrastructure and the number, diversity, and scale of reactive systems is expected to increase given the continuing proliferation of embedded, networked, and interactive systems. Decomposition and composition are two complementary techniques that are helpful when designing and implementing reactive systems, especially given such increases in complexity. We argue that the dominant techniques based on multiple sequential threads/processes thwart decomposition and composition and thus contribute to the accidental complexity associated with reactive system development.

Specifically, we believe that a model for reactive systems should facilitate *principled* composition and decomposition. Beyond defining units of composition and a means of composition, a model for reactive systems should facilitate practical techniques like recursive encapsulation and behavior abstraction through interfaces. Composition should be compositional meaning that the properties of a unit of composition can be stated in terms of the properties of its constituent units of composition. Finally, units of composition should be subject to substitutional equivalence meaning that the definition of a unit of composition can be substituted for its use and vice versa. Substitutional equivalence allows a complex system to be reduced to a single unit and a complex unit to be decomposed into a system of simpler units.

In Chapter 3, we presented the *reactive component* model for reactive systems. The reactive component model is based on models like UNITY and I/O Automata in that computation is carried out via a sequence of atomic state transitions selected non-deterministically. To these models, reactive components adds facilities for recursive encapsulation and interfaces that facilitate third-party composition via explicit binding. A reactive component is a set of state variables, atomic state transitions, and ports. The state variables of a reactive component may only be updated by the transitions associated with that component to ensure compositionality. Ports have passive and active sides. Push ports allow a transition in a component to synchronously induce a transition in another component while pull ports allow components to export and access values and state. An action is a transition with a precondition that can be executed by the scheduler. A reaction is a passive push port and transition that can be linked to an active push port to form an atomic transition that spans multiple components. A getter is a passive pull port and an expression that allows the state of a component to be accessed in a safe way. A transition is divided into two phases called the immutable phase and the mutable phase. Components may not change state in the immutable phase, which allows their state to be shared via temporary values. State is updated from the temporary values in the mutable phase of a transition. This division facilitates the composition of transitions in a principled way as it provides a clear interpretation of the state of a component before and after a transition. This, in turn, allows the properties derived from the state variables and transitions in one component to be linked to the properties of other components to facilitate compositional reasoning.

The interface-based composition semantics of reactive components introduces the possibility of non-deterministic transitions. A non-deterministic state transition occurs when a state variable is updated in disparate ways in different sub-transitions. The detection of non-deterministic state transitions arising from composition is generally undecidable. However, allowing a reactive component instance to serve as a proxy for its state variables reduces the problem of detecting non-deterministic transitions to simple graph and set theoretic problems.

Practicality was one of our goals when designing the reactive component model. That is, we intended it to be implemented and used to design and build real-world systems in an effort to reduce the accidental complexity associated with reasoning about a system using one set of semantics and implementing it using another. Thus, we presented the rc_{go} programming language for reactive components in Chapter 4. We adopted the Go programming language as the basis for rc_{go} and added syntax and semantics to support the elements of the reactive component model. The major challenge when designing the language was supporting reference and move semantics while preserving the isolation of state between reactive components. Reference semantics are important as they allow the construction of arbitrary linked data structures and move semantics are important for efficient communication between reactive components. Supporting these features required techniques from race-free programming languages. Thus, we added intrinsic and indirection mutability attributes to pointer types. Immutable indirection mutability is used to enforce the immutable phase of state transitions while foreign indirection mutability prevents pointers referring to a component's state from being saved by another component. To facilitate move semantics, we introduced a transferrable heap type which facilitates the construction and transfer of self-contained linked data structures.

Chapter 5 described the implementation of key features for the rc_{go} run-time system. The two major assumptions leveraged throughout the design of the run-time system are 1) components instances serve as a proxy for their state variables and 2) the reactive systems being implemented are static meaning that the number and configuration of the reactive components in the system are fixed. With these two assumptions we present an algorithm that checks for sound composition. The algorithm treats each transaction, i.e., a composed transition, as a graph with nodes corresponding to actions, reactions, activate statements, and push ports. This algorithm checks for non-deterministic transactions by ensuring that the graph contains no cycles and that a component instance participates in at most one non-empty activate statement.

The execution of a transaction requires two passes over a transaction graph corresponding to the immutable phase and mutable phase. The main challenge when computing the immutable phase is recording the context of each action/reaction that executes an activate statement so that its continuation may be executed in the mutable phase. To accomplish this, we created the novel *synchronized two-phase calling convention* which captures the context of an activate statement using an ordinary call stack. The significance of this approach is that transactions can be executed efficiently without allocating a stack frame on the heap.

The interpreter for rc_{go} uses garbage collection to reclaim memory. Garbage collection avoids dangling pointers which could threaten the isolation of state among component instances. All component state in rc_{go} may be attributed to exactly one reactive component instance.

This admits an embarrassingly parallel garbage collection algorithm, as garbage collection can be performed on each component in parallel.

The schedulers described in Chapter 6 are the most significant parts of the rc_{go} implementation. A scheduler has the responsibility of executing transactions with fairness. The challenge when designing and implementing a scheduler is to convert the logically serial execution of the reactive component model to physically concurrent execution. To do this, the scheduler utilizes the composition analysis to determine which transactions may be executed in parallel. We present two scheduler implementations: one is based on a globally shared work queue while the other is based on a static partitioning of the transactions. To evaluate the scheduler, we simulate a number of rounds in a simple request-response protocol. In the first design, the rc_{go} schedulers outperform a custom multi-threaded implementation due to excessive context switching in the multi-threaded program. This suggests that reactive components are viable as a concurrent event-based approach to reactive systems. The request-response protocol was then rewritten to be more conducive to the test hardware. The results of this experiment show that additional work is required to raise the performance of the rc_{go} interpreter to be on par with optimized libraries for multi-threading.

7.1 Conclusions

The reactive component model is viable for designing and implementing reactive systems. The main goal of this work is to reduce the accidental complexity associated with the design and implementation of reactive systems. We believe the source of this accidental complexity is the mismatch between the semantics of reactive systems and the sequential multi-threaded techniques used to implement them. Furthermore, we observe that sequential multi-threaded techniques fail to adequately manage complexity in the face of composition and decomposition. There are three main ideas in the reactive component model that make it a viable solution to these problems. First, the reactive component model addresses the semantics of reactive systems by interpreting computation in reactive systems as a non-deterministic sequence of atomic events which have the added benefit of composing well. Second, the reactive component model uses encapsulation to guarantee compositionality. The state of each reactive component instance may only be manipulated by the transitions of that component and therefore properties established from the transitions may never be violated through subsequent composition. This principle extends to constellations of interacting (composed) components that are themselves encapsulated. Third, compound state transitions spanning multiple components are formed using parallel composition instead of sequential composition. Parallel composition as expressed through the immutable phase and mutable phase concepts in the reactive component model provides a clear interpretation of compound atomic transactions.

The semantics of reactive components can be checked efficiently. The three main checks are 1) the enforcement of immutability during the immutable phase, 2) prevention of shared state, and 3) the detection of non-deterministic transitions. The first two properties are enforced through the type system and type checking algorithm. The detection of non-deterministic state transitions can only be performed once the set of concrete instances is known. The current algorithm leverages the static system assumption to create concrete transactions from compound transitions. These transactions can be checked in polynomial time. The key point is that the reactive component model does not rest on assumptions that cannot be realized in practice, as all of these algorithms have been implemented.

There is evidence that a run-time system for reactive components could be made efficient enough to compete with optimized multi-threaded approaches. A common approach to implementing high-performance reactive systems is to use a number of concurrent event loops which maximizes the use of available cores while minimizing context switches. The scheduler implementations in Chapter 6 are examples of this kind of design. The efficacy of this approach is seen in the first experiment where the reactive component schedulers outperform a multi-threaded implementation due to context switching. The second experiment demonstrates that additional work is necessary to improve the performance of the reactive component schedulers to make them comparable to custom multi-threaded implementations.

7.2 Future Work

This dissertation has presented the reactive component model as an alternative to sequential threads for designing and implementing reactive systems. As a model, sequential thread-based computation is firmly entrenched in many areas including hardware (synchronous sequential processors), operating systems (process and thread abstractions), and programming languages. Thus, we may consider possibilities such as hardware architectures specifically designed for reactive components and operating systems where the main abstraction is the reactive component or transaction instead of the thread. These topics are quite ambitious and additional work is required before these topics may be adequately addressed. First, we must generalize the reactive component model for dynamic systems (Section 7.2.1). This is necessary as an operating system must be able to load and configure reactive components and many reactive applications are naturally formulated using dynamic configurations. Second, we must consider the problem of scheduling transactions, as the scheduler will have a significant impact on the performance of reactive component applications (Section 7.2.2).

7.2.1 Dynamic Systems

Two assumptions were necessary to make the analysis of systems of reactive components tractable. First, we assume that a reactive component instance is a proxy for its state variables. Pull ports and getters reduce the impact of this assumption as a designer can decompose at will which yields more component instances for fine-grained analysis. Second, we assume that a reactive system has a fixed number of component instances in a fixed configuration. While we argue that many systems of interest have this property, removing or weakening this assumption would make the reactive component more general and more applicable to areas like enterprise level distributed systems and cloud computing.

The first level of support with regards to dynamic systems is adding support for dynamic sets of components and bindings. To illustrate, consider how the implementation of a TCP or UDP stack could use a dynamic array or set of components. The various ports in TCP/UDP represent independent data flows that may be processed in parallel. To exploit this parallelism in the reactive component model, each port should be processed by a unique reactive component instance. It is theoretically possible to declare a component instance for the 65,536 ports defined in the protocol. However, doing so in practice would be wasteful as only a subset of ports are active at any given time. Thus, we require the ability to dynamically allocate and bind component instances.

Models like UNITY and I/O Automata use parameterized models to reason about systems consisting of an arbitrary number of program instances. For the TCP/UDP port example, these models would assume that all 65,536 ports exist and are sent activate and deactivate messages as necessary. From an implementation perspective, it may be possible to use this idea and automatically allocate and deallocate resources for component instances when they receive activate and deactivate messages.

Providing support for dynamic sets of components and bindings requires further extensions to the model, language, and implementation. The major extensions to the model would include adding support for dynamic arrays or sets of components and parameterized bindings that allow a component to interact with a dynamically allocated component. The next step then would be mapping the extended semantics into the $\mathrm{rc}_{\mathrm{go}}$ programming language. Supporting dynamically allocated components and bindings in the implementation also would require extending the check for sound composition and providing run-time support for the features in the language. Checking for sound composition in a system with dynamic sets of components would require the run-time system to perform an inductive proof over all sets of dynamic components in the system. That is, the run-time system must prove that all transactions remain deterministic when any set of components increases in size.

The dynamic sets of components and bindings described so far are subject to static analysis as the dimensions of variability are expressed in the code. The next level of support would involved unplanned variability where new component types and instances are loaded and bound. Supporting this level of dynamism requires support for programmatic loading and binding including a service discovery mechanism that allows components to publish their existence and other components to find them. In terms of enforcing the reactive component model, the check for sound composition must be performed at runtime and constitutes a form of admission control. Thus, additional work is needed to determine if the sound composition check can be modified for use in an on-line setting or if additional restrictions must be placed on the model to ensure an efficient on-line check.

7.2.2 Scheduling

There are three main areas that may be addressed to improve the performance of the Instance and Partitioned schedulers. First, we may attempt to improve the efficiency of the schedulers through better design and implementation. The evaluation in Chapter 6 can serve as a basis for future improvements. Second, we may attempt to reduce the overhead of the Linux scheduler. In Linux, the threads made available via the pthreads library are scheduled by the Linux kernel. The Instance and Partitioned schedulers, then, are themselves scheduled by the Linux kernel. Thus, it may be necessary to implement a scheduler for reactive components at the kernel level to restrict this source of overhead. Third, we may move from interpretation to compilation. To test this idea, we timed a loop to sum the numbers in the range [0, 1,000,000) and averaged it over 1,000 runs for a C++ implementation and a reactive component implementation. The average time for the reactive component implementation was 0.1304896 seconds and the average time for the C++ implementation was 0.0009984909 seconds; a speed-up factor of 131. This is very much a "back of the envelope" calculation; the actual improvement in moving from interpretation to compilation may be less than this.

We selected the AsyncClock and SyncClock systems to evaluate the schedulers because they are small enough to understand but complex enough to show interesting behavior. The actual computation performed by these systems is minimal, i.e., setting Boolean flags and incrementing counters. Most likely, these operations can be performed in a single clock cycle (for compiled programs). One obvious direction for future work, then, is to experiment with complex transactions and varied workloads. However, it is important to consider the possibility that real workloads will contain computationally simple transactions such as those found in the clock systems. Furthermore, one of the motivations for decomposing systems is to break up complex systems into smaller, reusable, and easier to understand systems. The results in Chapter 6 demonstrate the diminishing return where the transactions become so simple that execution is dominated by overhead, i.e., it is more efficient to serialize execution than execute actions in parallel. Simple transactions are a foreseeable consequence of decomposition and designers should not be penalized for decomposing complex systems. Thus, one of the goals of scheduler design is to push the horizon of the diminishing return as far as possible. When coupled with responsiveness, the desired property in a scheduler is one whose throughput does not diminish with entanglement.

One goal for a reactive component scheduler may be to optimize the evaluation of preconditions. Two possible goals are to reduce the number of times a precondition is evaluated (lazy vs. eager) or to reduce the overhead of evaluating a single precondition. The reactive component model allows arbitrary Boolean expressions to be used as preconditions. One idea may be to eliminate preconditions and replace them with explicit scheduling instructions, but this may become tedious and error prone. A compromise solution may involve restricting the complexity of preconditions to simple Boolean expressions, i.e., with no function calls. With this restriction, it may become possible to determine which activate statements enable/disable a transaction.

To be safe, all concurrent schedulers require synchronization to protect the mutable state in a transaction. The approach taken by the oblivious Instance and Partitioned schedulers is to acquire locks protecting the state of each involved component before executing a transaction. The Partitioned scheduler demonstrates how asynchronous locking can be used to create a work-conserving scheduler. As indicated by the results in Chapter 6, a major challenge is to make synchronization as efficient as possible.

Another direction for future work is to explore options that accomplish synchronization without locking or with minimal locking. Partitioning combined with non-preemption may create opportunities to synchronize without locking. Let u be a transaction assigned to a specific scheduler thread and let I be the set of instances involved in u. Furthermore, let all transactions involving any component in I be mapped to the same scheduler thread (the single-threaded schedulers described in Chapter 6 do this by virtue of their design). In this scenario, no locks are needed for a safe execution of u because all other transactions that could change the relevant component state are excluded from executing due to the non-preemptive nature of the scheduler. This phenomenon is illustrated graphically by coloring the nodes in a race graph according to the thread to which they have been assigned. A transaction whose immediate neighbors all share the same color as the transaction itself can execute without acquiring locks. This suggests that algorithms that detect clusters of transactions in the race graph may be used to optimize the assignment of transactions to scheduler threads.

Locking also may be optimized by performing operations on the race graph. For example, adjacent transactions in the race graph can be merged and executed under the same set of locks. This procedure can be used to create complex transactions that amortize the locking overhead. Most likely, the merging algorithm would make use of some heuristic that measures the perceived benefit of merging the transactions. For example, the edges in the race graph could be weighted with the Jacquard Index of the instance sets, i.e., the edges are weighted with a score indicating that the set of locks are similar. The merging algorithm must balance the goal of simplifying the locking with the goal of exploiting concurrency. For example, merging the Request and Response transactions eliminates the potential concurrency between the Request and Tick transactions in the *AsyncClock* system.

Similarity between instance sets suggests that locks may be eliminated by combining them. To illustrate, the *AsyncClock* system contains three locks corresponding to the three components. Suppose that the same lock is used to protect both the Client and the Server. This reduces the number of locks needed to execute the Request transaction from two to one and the number of locks needed to execute the Response transaction from three to two.

The challenges associated with locking invite us to step back and consider the fundamental aspects of the problem that necessitate a locking protocol. The scheduler designs put forth so far are distributed in the sense that each scheduler thread is making an independent decision about the next transaction to execute. Locking is the means by which the scheduler thread discovers the decisions made by the other scheduler threads to determine if the transaction under consideration is compatible with the transactions already chosen by the other threads. The locking protocol becomes unnecessary if the decisions made by the other threads are already available, i.e., a knowledgeable scheduler. This suggests that orchestration may be used to ensure safety in a multi-threaded scheduler without locking. In this model, a manager thread assigns transactions to scheduler threads who execute the transactions without acquiring locks. The primary job of the manager thread is to enforce safety by only allowing the concurrent execution of disconnected transactions in the race graph. This approach may make use of a dedicated manager thread (Producer-Consumer) or use the Leader-Follower pattern [84]. An efficient implementation of the scheduler state, especially if it is shared through the Leader-Follower pattern, is a concern for this kind of scheduler. For systems with a fixed set of transactions, a compiler may be able to generate a scheduling automaton by enumerating the scheduler states S, pruning unsafe states and transitions according the rules outlined in Section 6.1, pruning transitions to ensure fairness, etc. This formulation also may have the advantage of yielding compact scheduler state.

Cache awareness and migration are two concerns that reactive component schedulers share with thread schedulers. Cache awareness attempts to improve the performance of a system by scheduling a computation on the same processor core because the state required for the computation may already be available in the cache. Thus, a reactive component scheduler may attempt to create an affinity between a transaction and a scheduler thread or processor core. Partitioning on the basis of shared component state satisfies this implicitly. Migration reassigns a computation to a different core to balance the load among the cores or free a core so that it may be shut down. The challenge with respect to migration is to define load imbalance in a meaningful way. The migration algorithm will most likely be expressed as an optimization problem that attempts to find a global optimum for the throughput and latency of each action.

Another design dimension for a reactive component scheduler is preemption. The main use of preemption is to share a physical core among many computations that are ready to execute. A reactive component scheduler may wish to preempt a long-running transaction to execute other enabled transactions. A preemptive scheduler for reactive components can take advantage of existing work on thread preemption.

7.3 Broader Impacts

Reactive systems have had a profound impact on society and will continue to impact society for the foreseeable future. Some reactive systems like the Internet and smart phones have high visibility while others, like the army of micro-controllers present in a modern automobile or a home appliance, are less conspicuous but nevertheless help us with our daily activities and contribute to our safety and comfort. Some reactive systems, like pace makers, life support machines, and robotic surgical instruments, even have a direct impact on our health and well being. The goal of this research is to ensure the quality and reliability of reactive systems in the face of predicted increases in size, diversity, and complexity.

Appendix A

Partition Tables

Symbol	System	Server	Counter	Client	Response	Tick	Request	Count
A	0	0	0	0	0	0	0	17
B	0	0	0	0	0	0	1	16
C	0	0	0	0	0	1	0	20
D	0	0	0	0	0	1	1	12
E	0	0	0	0	1	0	0	11
F	0	0	0	0	1	0	1	17
G	0	0	0	0	1	1	0	11
H	0	0	0	0	1	1	1	12
I	0	0	0	1	0	0	0	13
J	0	0	0	1	0	0	1	6
K	0	0	0	1	0	1	0	17
L	0	0	0	1	0	1	1	20
M	0	0	0	1	1	0	0	16
N	0	0	0	1	1	0	1	25
O	0	0	0	1	1	1	0	14
P	0	0	0	1	1	1	1	12
Q	0	0	1	0	0	0	0	14
R	0	0	1	0	0	0	1	11
S	0	0	1	0	0	1	0	9
T	0	0	1	0	0	1	1	15
U	0	0	1	0	1	0	0	20
V	0	0	1	0	1	0	1	12
W	0	0	1	0	1	1	0	18

Symbol	System	Server	Counter	Client	Response	Tick	Request	Count
X	0	0	1	0	1	1	1	16
Y	0	0	1	1	0	0	0	16
Z	0	0	1	1	0	0	1	16
a	0	0	1	1	0	1	0	18
b	0	0	1	1	0	1	1	21
c	0	0	1	1	1	0	0	16
d	0	0	1	1	1	0	1	9
e	0	0	1	1	1	1	0	14
f	0	0	1	1	1	1	1	11
g	0	1	0	0	0	0	0	16
h	0	1	0	0	0	0	1	17
i	0	1	0	0	0	1	0	12
j	0	1	0	0	0	1	1	20
k	0	1	0	0	1	0	0	16
l	0	1	0	0	1	0	1	16
m	0	1	0	0	1	1	0	13
n	0	1	0	0	1	1	1	17
o	0	1	0	1	0	0	0	25
p	0	1	0	1	0	0	1	22
q	0	1	0	1	0	1	0	13
r	0	1	0	1	0	1	1	14
s	0	1	0	1	1	0	0	10
t	0	1	0	1	1	0	1	11
u	0	1	0	1	1	1	0	27
v	0	1	0	1	1	1	1	16
w	0	1	1	0	0	0	0	12
x	0	1	1	0	0	0	1	18
y	0	1	1	0	0	1	0	17
z	0	1	1	0	0	1	1	12
0	0	1	1	0	1	0	0	17
1	0	1	1	0	1	0	1	7
2	0	1	1	0	1	1	0	11

Symbol	System	Server	Counter	Client	Response	Tick	Request	Count
3	0	1	1	0	1	1	1	17
4	0	1	1	1	0	0	0	19
5	0	1	1	1	0	0	1	19
6	0	1	1	1	0	1	0	16
7	0	1	1	1	0	1	1	17
8	0	1	1	1	1	0	0	21
9	0	1	1	1	1	0	1	22
@	0	1	1	1	1	1	0	22
$	0	1	1	1	1	1	1	13

Table A.1: Partitions for the AsyncClock system. The Symbol column contains the symbol used to represent this partition on plots. The System, Server, Counter, and Client columns indicate the thread used for the garbage collection action for the respective component. The Response, Tick, and Request columns indicate the thread used for the respective transaction. The Count column indicates the number of samples for this partition.

Symbol	System	Counter	Request	Tick	Count
A	0	0	0	0	117
B	0	0	0	1	137
C	0	0	1	0	119
D	0	0	1	1	114
E	0	1	0	0	117
F	0	1	0	1	140
G	0	1	1	0	135
H	0	1	1	1	121

Symbol	System	Counter	Request	Tick	Count

Table A.2: Partitions for the SyncClock system. The Symbol column contains the symbol used to represent this partition on plots. The System and Counter columns indicate the thread used for the garbage collection action for the respective component. The Request and Tick columns indicate the thread used for the respective transaction. The Count column indicates the number of samples for this partition.

References

[1] Clojure. `http://clojure.org/`.

[2] Go. `https://golang.org/`.

[3] Ioa language and toolset. `http://groups.csail.mit.edu/tds/ioa/`.

[4] *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, Pacific Grove, California, March 20-23, 1983*. Software engineering notes. Association for Computing Machinery, 1983.

[5] C++11. ISO/IEC 14882:2011, September 2011.

[6] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, March 2006.

[7] A. Adya, J. Howell, M. Theimer, W.J. Bolosky, and J.R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the 2002 Usenix ATC*, 2002.

[8] G.A. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[9] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys (CSUR)*, 15(1):3–43, 1983.

[10] J. Armstrong, R. Virding, C. Wikstr, M. Williams, et al. Concurrent programming in erlang. 1996.

[11] J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[12] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of java without data races. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 382–400, New York, NY, USA, 2000. ACM.

[13] J.C.M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2):131–146, 2005.

[14] D.W. Barron, J.N. Buxton, D.F. Hartley, E. Nixon, and C. Strachey. The main features of cpl. *The Computer Journal*, 6(2):134–143, 1963.

[15] J.A. Berstra and J.W. Klop. Fixed point semantics in process algebra. Technical Report IW 208, Mathematical Centre, Amsterdam, 1982.

[16] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[17] G. Booch. Object-oriented design. *ACM SIGAda Ada Letters*, 1(3):64–76, 1982.

[18] F.P. Brooks Jr. *The mythical man-month (anniversary ed.).* Addison-Wesley Longman Publishing Co., Inc., 1995.

[19] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 167–176, Washington, DC, USA, 1998. IEEE Computer Society.

[20] K.M. Chandy and J. Misra. *Parallel program design.* Reading, MA; Addison-Wesley Pub. Co. Inc., 1989.

[21] A. Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[22] W.D. Clinger. *Foundations of actor semantics.* PhD thesis, Massachusetts Institute of Technology, 1981.

[23] EF Codd, ES Lowry, E. McDonough, and CA Scalzi. Multiprogramming stretch: feasibility considerations. *Communications of the ACM*, 2(11):13–17, 1959.

[24] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, July 1963.

[25] F.J. Corbató, M. Merwin-Daggett, and R.C. Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 335–344. ACM, 1962.

[26] F.J. Corbató and V.A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 185–196. ACM, 1965.

[27] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux device drivers.* O'Reilly Media, 2005.

[28] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. Von Eicken, and K. Yelick. Parallel programming in split-c. In *Supercomputing'93. Proceedings*, pages 262–273. IEEE, 1993.

[29] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured programming.* Academic Press Ltd., 1972.

[30] M. Davis. *Computability & Unsolvability.* Dover Books on Computer Science Series. Dover, 1958.

[31] D.C. DeRoure. Parallel implementation of unity. *The PUMA and GENESIS Projects*, pages 67–75, 1991.

[32] E.W. Dijkstra. Cooperating sequential processes. Technical report, Technological University, Eindhoven, The Netherlands, September 1965.

[33] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

[34] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification.* 5.1 edition, June 2011.

[35] E. Emerson and E. Clarke. Characterizing correctness properties of parallel programs using fixpoints. *Automata, Languages and Programming*, pages 169–181, 1980.

[36] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, November 1976.

[37] Cormac Flanagan and Martín Abadi. Object types against races. In *Proceedings of the 10th International Conference on Concurrency Theory*, CONCUR '99, pages 288–303, London, UK, UK, 1999. Springer-Verlag.

[38] The Apache Software Foundation. `http://www.apache.org`.

[39] The Apache Software Foundation. `http://tomcat.apache.org`.

[40] D.P. Friedman and D.S. Wise. *The Impact of Applicative Programming on Multiprocessing.* Technical report (Indiana University, Bloomington. Computer Science Dept.). Indiana University, Computer Science Department, 1976.

[41] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows xp kernel crash analysis. *20th LISA*, pages 101–111, 2006.

[42] S.J. Garland, N.A. Lynch, J. Tauber, and M. Vaziri. *IOA user guide and reference manual.* Computer Science and Artificial Intelligence Labatory, 2003.

[43] C. Georgiou, N. Lynch, P. Mavrommatis, and J.A. Tauber. Automated implementation of complex distributed algorithms specified in the ioa language. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(2):153–171, 2009.

[44] K.J. Goldman. Distributed algorithm simulation using input/output automata. Technical report, DTIC Document, 1990.

[45] K. Gopinath and J.L. Hennessy. Copy elimination in functional languages. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 303–314. ACM, 1989.

[46] A. Granicz, D. Zimmerman, and J. Hickey. Rewriting UNITY. In Eobert Nieuwenhuis, editor, *Proceedings of the 4th International Conference on Rewriting Techniques and Applications (RTA 14)*, volume 2706 of *Lecture Notes in Computer Science*. Springer, June 2003.

[47] Dan Grossman. Type-safe multithreading in cyclone. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 13–25, New York, NY, USA, 2003. ACM.

[48] S. Halloway. *Programming Clojure*. Pragmatic Bookshelf, 2009.

[49] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[50] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[51] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM, 2001.

[52] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[53] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 271–285, New York, NY, USA, 1991. ACM.

[54] M. Huber. Maspar unity version 1.0. `ftp://sanfrancisco.ira.uka.de/pub/maspar/maspar_unity.tar.Z`, 1992.

[55] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983.

[56] G. Kahn. The semantics of a simple language for parallel programming. *proceedings of IFIP Congress74*, 74:471–475, 1974.

[57] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 105–112, New York, NY, USA, 1986. ACM.

[58] Shriram Krishnamurthi and Jan Vitek. The real software crisis: Repeatability as a core value. *Commun. ACM*, 58(3):34–36, February 2015.

[59] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[60] J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *ACM SIGPLAN Notices*, volume 23, pages 24–31. ACM, 1988.

[61] D. Lea. *Concurrent programming in Java: design principles and patterns*. Prentice Hall, 2000.

[62] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[63] E.A. Lee. Heterogeneous actor modeling. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 3–12. ACM, 2011.

[64] N.A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[65] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 141–154. ACM, 1983.

[66] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*, volume 1. Springer, 1992.

[67] Jules Mersel. Program interrupt on the univac scientific computer. In *Papers presented at the February 7-9, 1956, joint ACM-AIEE-IRE western computer conference*, AIEE-IRE '56 (Western), pages 52–53, New York, NY, USA, 1956. ACM.

[68] R. Milner. *A calculus of communicating systems*. Springer-Verlag New York, Inc., 1982.

[69] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge Univ Pr, 1999.

[70] B. Murphy. Automating software failure reporting. *Queue*, 2(8):42–48, 2004.

[71] C. Okasaki. *Purely functional data structures*. Cambridge Univ Pr, 1999.

[72] IFIP Working Group 2.1 on Algol, S.A. Schuman, and Institut de recherche d'informatique et d'automatique. *New Directions in Algorithmic Languages*. Institut de recherche d'informatique et d'automatique., 1975.

[73] J. Ousterhout. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, 1996.

[74] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 116–127, New York, NY, USA, 1992. ACM.

[75] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[76] R.L. Patrick. General motors/north american monitor for the ibm 704 computer. 1987.

[77] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981.

[78] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

[79] S. Radha and C.R. Muthukrishnan. A portable implementation of unity on von neumann machines. *Computer Languages*, 18(1):17–30, 1993.

[80] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[81] James Reinders. Transactional synchronization in haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`, February 2012.

[82] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[83] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 275–288. ACM, 2009.

[84] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-oriented software architecture: Patterns for concurrent and networked objects*, volume 2. Wiley, 2000.

[85] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[86] A. Silberschatz, P.B. Galvin, and G Gagne. *Operating system concepts*, volume 7. Addison-Wesley, 2005.

[87] A. Silberschatz, P.B. Galvin, G. Gagne, and A. Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley, 1998.

[88] Lambert M. Surhone, Mariam T. Tennoe, and Susan F. Henssonow. *Node.Js*. Betascript Publishing, Mauritius, 2010.

[89] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[90] Robert Endre Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

[91] J.A. Tauber. *Verifiable compilation of i/o automata without global synchronization*. PhD thesis, Massachusetts Institute of Technology, 2004.

[92] J.A. Tauber, N.A. Lynch, and M.J. Tsai. Compiling ioa without global synchronization. In *Network Computing and Applications, 2004.(NCA 2004). Proceedings. Third IEEE International Symposium on*, pages 121–130. IEEE, 2004.

[93] M.J. Tsai. Code generation for the ioa language. Master's thesis, Massachusetts Institute of Technology, 2002.

[94] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[95] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[96] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78. ACM, 1990.

[97] W. Walker and H.G. Cragon. Interrupt processing in concurrent processors. *Computer*, 28(6):36–46, 1995.

[98] M.V.W.D.J. Wheeler and S. Gill. The preparation of programs for an electronic digital computer, 1951.

[99] Anthony Williams. *C++ concurrency in action practical multithreading*. Manning, Shelter Island, NY, 2012.

[100] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

Vita

Justin Wilson

Degrees	B.S. Summa Cum Laude, Electrical Engineering, December 2006 B.S. Summa Cum Laude, Computer Engineering, December 2006 Ph.D. Computer Science, December 2016

Publications

Wilson, J., Dai, M., Jakupovic, E., Watson, S., & Meng, F. (2007). Supercomputing with toys: harnessing the power of NVIDIA 8800GTX and playstation 3 for bioinformatics problems. In *Computational Systems Bioinformatics Conference* (Vol. 6, pp. 387-390).

Thomas, L., Wilson, J., Roman, G. C., & Gill, C. (2009, November). Achieving coordination through dynamic construction of open workflows. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing* (pp. 268-287). Springer Berlin Heidelberg.

Xi, S., Wilson, J., Lu, C., & Gill, C. (2011, October). RT-Xen: towards real-time hypervisor scheduling in xen. In *IEEE International Conference on Embedded Software (EMSOFT)* (pp. 39-48).

December 2016