A Model and Platform for Designing and Implementing Reactive Systems

Justin R. Wilson

Chapter 1

Introduction

Definitions. Manna and Pnueli classify programs as either being *transformational* or *reactive* [40]. As implied by their name, transformational programs transform a finite input sequence into a finite output sequence. Transformational programs can often be divided into three distinct phases corresponding to the activities of input, processing, and output. A compiler is a classic example of a transformational program as it transforms a source file into an object file. Formal models of computation like the Turing machine [53] and λ-calculus [15] are concerned with transformational programs.

In contrast, a reactive program is characterized by "ongoing interactions with its environment[40]." Whereas transformational programs are designed to halt and produce an output (when possible, i.e., the halting problem), reactive programs are often designed to run forever. The activities of input, processing, and output are thus overlapped in the execution of a reactive program. A web server is a classic example of a reactive program as it continually 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 [41], the Algebra of Communicating Processes [10], Cooperating Sequential Processes [21], Communicating Sequential Processes [33], the Actor Model [31][16][5], UNITY [14], and I/O automata [39].

Asynchrony and concurrency are two fundamental characteristics of reactive systems that make them difficult to develop. A reactive program and its external environment may evolve concurrently and must share common resources to facilitate communication (input and output). For example, in a system consisting of two threads that communicate using shared variables, each thread is in the environment of the other thread. Many of the difficulties associated with concurrency stem from the concurrent use of shared resources, e.g., the threads' shared variables. To ensure correctness in models that allow the environment to deliver input to a reactive program asynchronously, the environment must be viewed as an adversary that will deliver inputs at inopportune times. Since

a reactive program may run forever, the designers of reactive programs thus must often reason about infinite event sequences containing interleaved input, output, and processing.

Failure to account for asynchrony and concurrency results 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., a new processor, a new operating system, or part of the program takes more or less time than normal. Operating systems are particularly susceptible to timing bugs caused by device drivers [48]. Timing bugs can escape even a rigorous software development process and lay dormant for years [38]. 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) [2]. Timing bugs cost man-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 it reboots.

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., top-down design [55]. Composition, building a complex system from a number of simpler systems, is often used when *implementing* a system. The simpler systems are implemented, tested, and integrated to create larger systems [13]. 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, one goal is the ability to guide a design toward 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 [44]. Thus, decomposition is a significant aid to understanding complex systems.

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 deploying networks of sensors and actuators to monitor, control, and coordinate critical infrastructures such as power grids and telecommunication networks. These advances have also led to platforms targeted at the individual, such as smart phones, e-readers, and tablets. Applications for these personal platforms are necessarily interactive and therefore reactive. The leveling of processor speeds and the resulting trend toward multicore processors is also creating demand for reactive systems as it is believed that increases in performance must come through increasing concurrent applications [52].

A general trend toward distribution is also increasing demand for reactive

systems. Increasingly, network services form the core or are a critical component of an application and are fundamental to delivering content (e.g., downloading books and movies) and communication (e.g., 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 as networked systems, e.g., file servers, print servers, web servers, application servers, databases, etc., 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 reactive systems 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 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 simultaneously uses a service on the Internet, interacts with the user, and uses a variety of devices (e.g., an accelerometer) typically found in embedded environments. Similarly, sophisticated servers like web servers and databases are often built from collections of interacting reactive modules.

Challenges. The first challenge towards adequately supporting such 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 [13]." 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 [38]. 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 a platform that provides direct support for reactive semantics.

The second challenge is to ensure that reactive systems can be decomposed and composed in a principled way. Asynchrony and concurrency, two inherent characteristics of reactive systems, undermine decomposition and composition when not properly encapsulated and therefore limit our ability to design and implement complex reactive systems. Such problems of asynchrony and 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.

Limitations of the state of the art. State is fundamental to reactive systems and may be modeled directly (variables and assignment) or indirectly. Some examples of indirect approaches to modeling state include monads [54] which are typically used in functional languages such as Haskell, and a mail queue with message-behavior pairs [5] which is used in actor-oriented languages such as Erlang. The dominant languages used to implement reactive systems, such as C, C++, and Java, support the imperative programming paradigm by way of the von Neumann architecture and model state directly with variables and assignment. As we are interested in expressing reactive semantics directly, we limit the remaining discussion to languages like C, C++, and Java.

Imperative programming languages have three characteristics that are germane to reactive programs. The first characteristic is *deterministic sequencing*. Imperative programs are constructed by concatenating assignment statements and control statements. Control flows implicitly to the next statement unless modified by a control statement (conditional or loop). In an imperative program, the next statement to be executed is *always* determined by the program. The second characteristic is *sequential composition*. In the imperative programming paradigm, a block of statements can be packaged as a subroutine or function. This facilitates composition since complex logic can be built from smaller units. The composition is sequential since the same sequential flow of control applies to the change in control introduced by the call. The third characteristic is *explicit atomicity* meaning that the developer is responsible for designating which sequences of statements appear as a single statement with respect to another program in the environment. Of interest to this research is the combined effect of deterministic sequencing, sequential composition, and explicit atomicity on the accidental complexity of reactive programs and their composition and decomposition.

A key limitation of deterministic sequencing and explicit atomicity is the burden it places on the developer. The misuse of synchronization primitives, which becomes likely as transitions become complex, may introduce concurrency hazards, e.g., deadlock [21], which manifest themselves as timing bugs. A number of design patterns for concurrency have been developed to help developers avoid concurrency hazards [49], [37]. These design patterns represent a move toward implicit atomicity as they often attempt to leverage language features to control atomicity, e.g., scoped locking [49]. While some of these patterns have been incorporated into programming languages, e.g., the `synchronized` keyword of Java implements the thread-safe interface pattern [49], they are most

often enforced by convention and therefore easily violated or ignored (e.g., if a new developer is unaware of the convention). In practice, the use of synchronization primitives has proven to be tedious and error prone [52].

Correct synchronization with sequential composition and explicit atomicity is a holistic problem that resists encapsulation. Reactive programs based on sequential composition are often designed using modular principles such as object-oriented programming. The sequence of transitions that constitute a reactive program 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 against 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 fragile, i.e., modifications tend to introduce new bugs.

Reactive programs and processes based on deterministic sequencing and explicit atomicity are not subject to principled decomposition and composition. To illustrate, consider three reactive programs A, X, and Y that are based on deterministic sequencing 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 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 taking a transition in one of the reactive programs. 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 sequence. Thus, reactive programs based on deterministic sequencing are not subject to recursive encapsulation and substitutional equivalence.

TODO Approach. To reduce the accidental complexity associated with the design and implementation of reactive systems while ensuring that reactive systems can be decomposed and composed in a principled way, we developed a new model and platform for reactive systems based on the direct manipulation of state, implicit atomicity, and non-deterministic sequencing. Implicit atomicity relieves the burden of specifying critical sections with synchronization primitives, which seems to be one of the greatest sources of accidental complexity in reactive systems, while the semantics of non-deterministic sequencing allows reactive programs to be subject to principled decomposition and composition.

TODO Contributions. This research makes three contributions to the state of the art in reactive system development. In section 3, we develop a model for reactive systems based on implicit atomicity and non-deterministic sequencing

in an effort to reduce the accidental complexity associated with their development and enable principled decomposition and composition. In section 4, we propose a programming language based on the model and the assumption of a static system. In section 6, we propose a qualitative evaluation of the model to determine if the model facilitates design processes based on principled composition and decomposition and a quantitative evaluation of its implementation to determine if the model has an efficient implementation on modern architectures.

Chapter 2

Background and Related Work

Principles of composition and decomposition. 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 definable 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.

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 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[8]." The state of a reactive system is often captured in program variables, e.g., Dijkstra's Cooperating Sequential Processes [21], messages in a channel or queue, e.g., Milner's Calculus of Communicating Systems [41], the Actor Model [5], or some combination of the two, e.g., Kahn Process Networks [35]. For generality, platforms typically allow complex state to be composed from primitive state, e.g., arrays, records, tuples, lists, sets, etc.

Computation in a reactive system then can be viewed as a sequence of state transitions [45]. As these transitions may be complex, platforms typically allow complex state transitions to be composed from primitive state transitions. 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 [9]. 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 object-oriented principles where updates to shared state are exposed in the form of structured transitions as opposed to raw assignment, e.g., a method to place a message in a queue. Synchronization and communication are identified as a necessary activities in a reactive system [6]. The two approaches to communication are *shared variables* and *message passing* [6].

Multiple reactive processes can effect state transitions which may overlap in time, i.e., *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., [39], [14], [40].

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 [21] and Hoare's Communicating Sequential Processes [33]. 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 [14] and Lynch's I/O automata [39].

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-test, compare-and-swap, that may atomically update state and/or alter the flow of control [6]. Synchronization primitives can be used to construct more general synchronization mechanisms like semaphores and monitors [21]. The goal of synchronization is to create atomic sequences of transitions called *critical regions* or *critical sections* [6]. 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 [6]. 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 [21]. Andrews and Schneider describe a number of techniques associated with this approach including coroutines, fork/join, spin locks, semaphores, conditional critical regions, and monitors [6]. This model is supported by widely available platforms, i.e., operating systems, via processes with shared memory or threads [51]. A number of design patterns have been developed based on this approach, e.g., [49], [37]. As described by Lee [38], 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 [11], Split-C [18], C++11 [3]. Sutter and Larus [52] and Lee [38] provide a modern perspective on the difficulties associated with this approach. In [52], 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 proposed work is a step in this direction.

Atomic transactions have been proposed as an alternative to locks when synchronizing multiple processes. The idea of an atomic transaction originated in the field of databases [23]. Knight [36] and then Herlihy and Moss [30] proposed cache-based hardware support for atomic transactions. Hardware atomic transaction are forthcoming on modern processors [47]. Software transactional memory, proposed by Shavit and Touitou [50], has sparked a great deal of interest and have been implemented in a number of languages, e.g., Clojure [29].

Another popular approach to reactive systems is messaging passing with deterministic sequencing as is done in Hoare's Communicating Sequential Pro-

cesses [33]. This model is also supported by widely available platforms via pipes, message queues, and sockets [51]. A message passing channel may either be a fixed size or unbounded and accessed synchronously or asynchronously by either the sender or the receiver [6].

Events, as proposed by Ousterhout [43], and embodied in the Reactor and Proactor design patterns [49], are 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 systems 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[4]." 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.

Message passing may also be paired with non-deterministic sequencing. In the asynchronous mode, the action of sending is distinct from receiving and requires a mail queue to buffer the unprocessed messages. This is the approach taken in Hewitt's Actor model [31] which is gaining popularity as exemplified by the Erlang programming language [7]. The proposed research will attempt to explore the synchronous mode where sending and receiving are combined in one atomic event which creates opportunities for synchronous composition.

UNITY and I/O automata. The UNITY model of Chandy and Misra [14] and the I/O automata model of Lynch [39] are two influential models for reactive systems based on non-deterministic sequencing. Execution in these models proceeds with a scheduler repeatedly selecting a transition, evaluating its guard, and executing it if the guard is true. The scheduler is assumed to be fair meaning that it will select (but not necessary 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 (named-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 of the sizes of the two constituent programs as opposed to the product.

Superposition is property-preserving while union is not property-preserving. However, superposition has certain weaknesses as described by the authors of UNITY [14]:

> 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 say 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 the language support for doing so, the I/O automata [39] 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 a guard and may be selected and executed by the scheduler. Input actions 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 I/O automata is property-preserving.

Of interest to the proposed work is the implementation of these models 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 [28]. Other initiatives to implement the UNITY programming language are summarized in [28]:

> Few compilers have been developed for the UNITY language. DeRoure's parallel implementation of UNITY [20] compiles UNITY to a common backend language, BSP-occam; Huber's MasPar UNITY [34] 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 [46][1].

Goldman's Spectrum Simulation System [26] 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 [1]. The IOA toolkit does

[1] We believe all of these projects, including [28], are now defunct.

contain a source-to-source compiler (IOA to Java) and a run-time that has been used to compile and execute distributed protocols [25].

Chapter 3

Model

In this chapter, we present a model for reactive programs called *reactive components*. The model is biased toward practical software development as opposed to formal methods. Consequently, the model favors utility, practicality, flexibility, and ease of implementation. The model is presented as a sequence of modifications to the I/O automata model of Lynch [39].

As a starting point, we consider the *Clock* automaton presented in [39]. A *Clock* represents a free-running timer that can be sampled. The *Clock* automaton has an internal action *tick* that increments *counter*, an input action *request* that sets $flag$, and an output action $clock(t)$ that sends the current state of *counter* and clears $flag$. The precondition of *tick* is true and the precondition of $clock(t)$ is $flag = true$. A simple *ClockClient* contains an output action *request* for making a request which sets $flag$ and an input action $clock(t)$ for receiving the response which clears $flag$. The precondition of *request* is $flag = false$. Note that the $flag$ in *Clock* and the $flag$ in *ClockClient* are distinct variables.

Figure 3.1 shows a graphical representation of the composition of *Clock* and *ClockClient*.

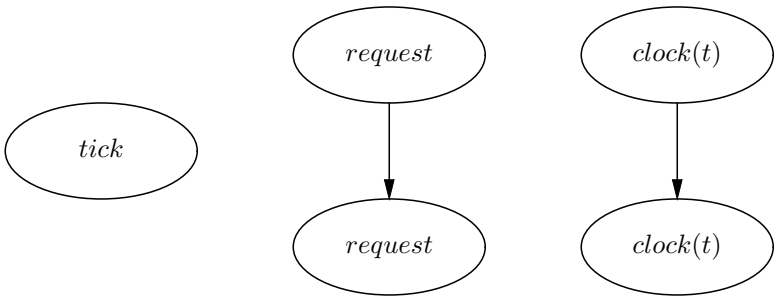

Figure 3.1: Graphical representation of the composition of a *Clock* and *ClockClient*

The main limitation of the I/O automata model is that it only permits the pair-wise linking of transitions. To illustrate, we define an α-statement to be a transition that can be selected by the scheduler and a β-statement to be a transition that cannot be selected by the scheduler but must be linked to another transition to be executed. α-statements correspond to local actions in the I/O automata model and stand-alone assignment statements in UNITY while β-statements correspond to input actions in the I/O automata model and superimposed clauses in UNITY. (Internal actions correspond to α-statements to which β-statements cannot be linked.) A single I/O automaton, then, can be viewed as a disconnected graph where each node is either an α-statement or a β-statement. A system composed of multiple I/O automata results in a forest of two-level trees where each root node is an α-statement and each child node is a β-statement.

A model for reactive systems is a set of definitions regarding their structure and behavior. A proposal for a new model for reactive systems should declare what is considered to be fundamental or desirable in a reactive system since this shapes the features of the model and makes the model appropriate for some systems (and perhaps inappropriate for others). The motivation behind our model, described in chapter 1, is the desire to express the semantics of reactive systems directly and structure systems through composition and decomposition. The definitions provided by a model for reactive programs must be rigorously defined to allow one to make precise statements about reactive programs and their behavior. The definitions are necessary for both informal, e.g., debugging, and formal, e.g., proving correctness, software development settings. This section describes our approach to rigorously defining the model. As formal verification is not the focus of this research, we leave the development of a proof logic for future work.

Limitations of existing models. UNITY [14] and I/O automata [39] are two formal models for reactive systems based on non-deterministic sequencing. Execution in these models proceeds with a scheduler repeatedly selecting a transition, evaluating its guard, and executing it if the guard is true. The scheduler is assumed to be fair meaning that it will select (but not necessary 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 (named-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 assign-

ment statements) is on the order of the sum of the sizes of the two constituent programs as opposed to the product.

Superposition is property-preserving while union is not property-preserving. However, superposition has certain weaknesses as described by the authors of UNITY [14]:

> 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 say 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 the language support for doing so, the I/O automata [39] 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 a guard and may be selected and executed by the scheduler. Input actions 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 I/O automata is property-preserving.

The main limitation of the I/O automata model is that it only permits the pair-wise linking of transitions. To illustrate, we define an α-statement to be a transition that can be selected by the scheduler and a β-statement to be a transition that cannot be selected by the scheduler but must be linked to another transition to be executed. α-statements correspond to local actions in the I/O automata model and stand-alone assignment statements in UNITY while β-statements correspond to input actions in the I/O automata model and superimposed clauses in UNITY. (Internal actions correspond to α-statements to which β-statements cannot be linked.) A single I/O automaton, then, can be viewed as a disconnected graph where each node is either an α-statement or a β-statement. A system composed of multiple I/O automata results in a forest of two-level trees where each root node is an α-statement and each child node is a β-statement.

Contribution. We propose a model for reactive programs that resolves the weakness of superposition and the structural limitations of composition in I/O automata. The idea for the proposed model is straightforward: extend composition to support general transition structures such as trees of arbitrary depth. A program in our model consists of state variables, α-statements, β-statements, and ports that allow transitions in different programs to be linked. Complex state transitions, i.e., those that involving a number of programs, can be constructed in a principled fashion by cascading β-statements, i.e., parallel composition. For example, a layered protocol can be expressed as a chain of transitions where the first transition corresponds to receiving a bit, the second transition corresponds to receiving a byte, the third transition corresponds to receiving a message, etc. Upon reception of the last bit, the entire chain will fire. Our first contribution will be a formal definition of the model and justification that it supports principled decomposition.

Formalization. A model for reactive systems should provide a rigorous set of definitions concerning units of composition and composition operations towards developing a platform that realizes the model. For a state-based model, the definitions should define the basic structure of a unit of composition and the encoding of state transformations. Similarly, the definitions should provide execution semantics that describe how state transformations in units of composition occur over time. The definitions of composition operations describe how units of composition are composed into systems and how systems can be interpreted with respect to the execution semantics. A formal definition will allow us to verify the claim that our model does indeed support principled composition and decomposition and defines the correctness criteria for a platform implementing the model. Furthermore, rigorous definitions are necessary when communicating about a design or analyzing the behavior of an existing system and are the first step toward developing a proof logic that allows one to establish the properties of a reactive system.

Units of composition. A unit of composition in our model is called a *reactive component*. A reactive component consists of a set of state variables, a set of input ports, a set of output ports, a set of internal ports, a set of member components, a set of α-statements, a set of β-statements, and a set of static port definitions called γ-statements. The *readable ports* in a reactive component comprise the input ports, internal ports, and the output ports of its members. The *readable objects* (those that can appear on the RHS of an assignment or port definition) in a reactive component comprise its readable ports and its state variables. Similarly, the *writable ports* in a reactive component comprise the output ports, internal ports, and the input ports of its members and the *writable objects* (those that can appear on the LHS of an assignment or port definition) in a reactive component comprise the writable ports and state variables.

An α-statement consists of a precondition and an effect. A precondition is a Boolean expression over a subset of the readable objects. An effect is a set

of assignment statements and port definitions. An assignment statement (:=) defines the next value for a state variable. A port definition (=) defines the value associated with a port. A β-statement consists of a trigger and an effect. A trigger is a port in the set of readable ports that when defined causes the application of the effect.

A port is either *static* or *dynamic* based on its usage. A static port is defined completely in terms of state variables and other static ports. A dynamic port is defined within the context of an α-statement. Ports referenced in a precondition must be static while triggers must be dynamic.

Based on our goal of principled composition and decomposition, reactive components are based on non-deterministic sequencing. The scheduler repeatedly (and non-deterministically) selects an α-statement, evaluates its precondition, and applies the effect if the precondition is true. Conceptually, applying the effect has two phases corresponding to port definition and state assignment. The port definition phase starts by defining ports using the effect of the α-statement. These port definitions can cascade by triggering β-statements and γ-statements. Any state assignment in the effect of a triggered β-statement is evaluated in the state assignment phase. In the state assignment phase, the RHS of each assignment statement is computed and then assigned to the state variable on the LHS. The scheduler is assumed to be fair meaning that it selects each α-statement an infinite number of times in an infinite execution. Interpreting a program as a set of assignment statements as opposed to a list facilitates the substitutional equivalence necessary for principled composition and decomposition.

To illustrate reactive components, we present a number of reactive components based on the Clock automaton of [39] encoded as a reactive component and listed in figure 3.2. The Clock component contains two state variables `counter` and `flag` indicating the current count and that the count has been requested, respectively. The Clock component contains an input port `request` and output port `clock` which can be used to request and receive the count. The use of `request` as a trigger and the definition of `clock` in the `Clock` α-statement implies that both ports are dynamic. The `Tick` α-statement increments the counter and is always enabled, i.e., its precondition is always true. The `Request` β-statement sets the `flag` when the `request` port is defined due to the execution of an α-statement. The `Clock` α-statement reports the current count by defining the `counter` port and resetting the `flag`. The effect of the `Clock` α-statement is atomic, i.e., the port definition and assignment statement are composed in parallel.

Figure 3.3 lists the component we use to poll the Clock component of figure 3.2. The ClockPoll component sends requests using the dynamic `request` port in the α-statement labeled `Request`. The ClockPoll component expects to receive the clock value on the dynamic `clock_in` port. The ClockPoll component has a γ-statement `G1` that replicates the `clock_in` port on the `clock_out` port when the `clock_in` port is even. Thus, ports can be conditionally defined and used to craft even more sophisticated transitions. The `clock_out` port is dynamic since it is derived from the `clock_in` port.

```
component Clock {
  var int counter (0)
  var bool flag (false)
  in signal request
  out int clock

  Tick: alpha (true) {
    counter := counter + 1
  }

  Request: beta (request) {
    flag := true
  }

  Clock: alpha (flag) {
    clock = counter
    flag := false
  }
}
```

Figure 3.2: Definition of the Clock component

```
component ClockPoll {
  out signal request
  in int clock_in
  out int clock_out

  Request: alpha (true) {
    request = defined
  }

  G1: if (clock_in % 2 == 0) {
    clock_out = clock_in
  }
}
```

Figure 3.3: Definition of the ClockPoll component

```
component ClockCounter {
  in int clock
  var int c (0)
  out int count

  Clock: beta (clock) {
    c := c + 1
  }

  G1: count = c
}
```

Figure 3.4: Definition of the ClockCounter component

```
component ClockSys {
  member Clock clock
  member ClockPoll poll
  member ClockCounter counter

  clock.request = poll.request
  poll.clock_in = c.clock
  counter.clock = poll.clock_out
}
```

Figure 3.5: Definition of a top-level component composing the Clock component, ClockPoll component, and ClockCounter component

Figure 3.4 lists the component we use to count responses from the ClockPoll component of figure 3.3. The ClockCounter component contains a single β-statement that increments the number of responses. The `G1` γ-statement defines the `count` output port to be the number of responses. The `count` port thus is *static* since it is derived solely from state variables.

Composition. Composition is achieved by declaring member components and establishing relationships among input and output ports through α-statements, β-statements, and port definitions. Using the Clock component of figure 3.2, the ClockPoll component of figure 3.3, and ClockCounter component of figure 3.4, we define a component that is a composition of these components. The resulting ClockSys component is listed in figure 3.5. The ClockSys component declares three members representing a clock component called `clock`, a component to poll the clock called `poll`, and a component to count even responses called `counter`. The various γ-statements associate the input and output ports of the member components. Figure 3.6 contains a block diagram representation of the ClockSys component that shows the port connections and directions.

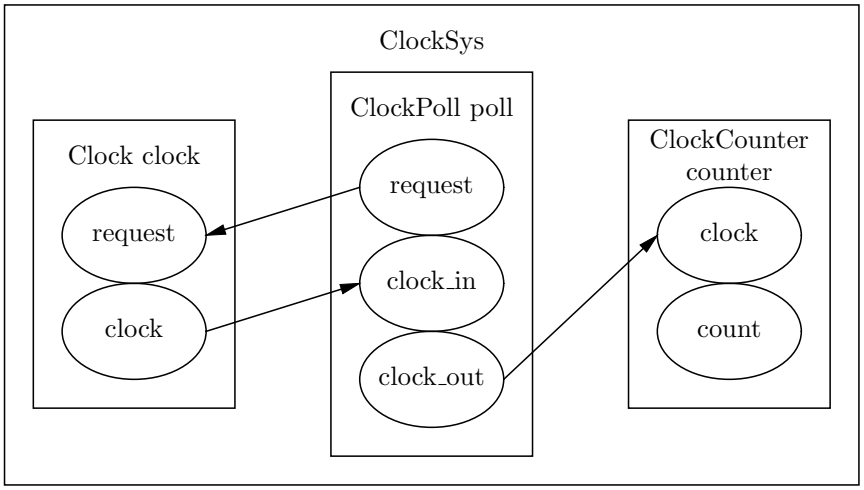

Figure 3.6: A block diagram representation of the ClockSys component

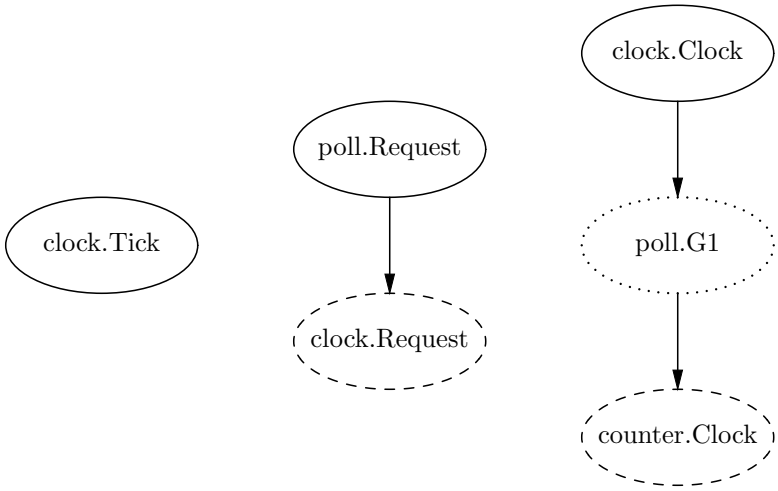

Figure 3.7: The α-forest for the ClockSys component. Solid nodes represent α-statements, dashed nodes represent β-statements, and dotted nodes represent γ-statements.

A component can also be interpreted as a forest of α-trees called an α-forest. Figure 3.7 shows the α-forest for the ClockSys component. A component instance can appear multiple times in an α-tree. Each α-tree defines a context consisting of a set of readable objects (LHS) and writable objects (RHS). An α-tree is well-defined if 1) each port in the set of readable objects is defined to a unique value and 2) there is a unique next state for each state variable in the set of writable objects.

The ability for a component instance to appear multiple times creates the opportunity for the composition of well-defined components to result in a component that is not well-defined. This situation might occur, for example, if a component simultaneously drives two inputs that are mutually exclusive. More subtly, one can envision creating a "feedback loop" that includes incompatible statements in the same component. By strengthening the well-definedness property to say that a port or next state can only be *defined* once in the context of an α-statement (instead of being defined to a unique value), one can mechanically check that a composed component is well-defined.

Principled composition and decomposition. The formalization of the model should allow us to show that the model supports principled composition and decomposition. The trivial principles of principled composition and decomposition come directly from the definition of the model.

To show that the behavior of a component is encapsulated by its interface, we first define its interface to be the set of input and output ports. The behavior of a component can then be described using execution traces where the events correspond to patterns of port definition. This treatment is consistent with trace properties in I/O automata which also serve to encapsulate and abstract the behavior of I/O automata.

To show that properties survive composition, we must prove theorems that correspond to the superposition theorem in UNITY and composition theorems of I/O automata. In general, there are two kinds of properties: those that are assertions about state variables and those that are assertions about execution traces. For state-based properties, we will replicate the superposition theorem of UNITY which is based on the observation that state variables cannot be modified outside of the component in which they are defined, therefore, all properties continue to hold when composed. For trace-based properties, we will replicate the composition theorems of I/O automata which state that traces compose due to the fairness property of the scheduler.

Substitutional equivalence can be shown through a constructive proof that substitutes the definition of a member for its instance. This can be performed mechanically by first changing all ports to internal and then renaming entities in the member component, e.g., prefixing with the name of the member. Substitution is linear in the size of the member due to the definition of a program as a set of assignment statements as opposed to a list. The compositionality of components is justified by substitutional equivalence.

Summary. In this section, we have proposed two contributions toward a model for reactive system development. First, we proposed formal definition for system structure that facilitates composition and decomposition. Second, we proposed to show that the model supports principled composition and decomposition by appealing to the definitions of the model.

Chapter 4

Implementation

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

An implementation of the model described in section 3 is necessary for at least three reasons. First and foremost, an implementation tests the practicality of the model. The act of implementing the model will reveal 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. 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.

Second, language support for a 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 [19] and object-oriented programming [12]. 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 a model raises the level of abstraction and allows reasoning about a system 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, e.g., checking by a compiler.

Third, 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 two different models, we can gain insight into the strengths and weaknesses of

the model. These ideas will be explored further in section 6.

Approach. Our approach to implementing the model in section 3 is based on a virtual machine called an α-machine that is designed to execute α-statements. The development workflow, then, consists of three steps. First, the developer defines a reactive system as a set of components in source files using a high-level language similar to the one used in the listings of section 3. Then the developer designates a top-level component that represents the system and compiles the source files to produce an α-machine image. Finally, the developer runs the system by loading an α-machine with the image created in the previous step.

This approach has two desirable characteristics. First, the virtual machine facilitates a separation of concerns between syntax and operational semantics. The issues of compilation and execution can be divided by the interface provided by the virtual machine. Thus, we can focus on implementing an α-machine without considering the high-level language used to encode component definitions. Similarly, we can focus on the translation of a high-level language for component definitions to the language of the virtual machine without considering the details of how the virtual machine is implemented. Second, writing a compiler for a virtual machine is often easier than writing a compiler for a physical machine. The target provided by a virtual machine is often at a higher-level of abstraction than a physical machine and can contain features and instructions that simplify or optimize certain tasks. A design based on a virtual machine will allow us to focus on the semantics of reactive programs instead of the complexities of processor architecture.

Static system assumption. To make implementation tractable, we will assume that the systems to be implemented have a static topology meaning that all reactive components are statically allocated. 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 equivalent to systems that assume a fixed number of actors and is roughly equivalent to systems that assume a fixed number of threads. These assumptions are common in embedded and real-time systems due the combination of limited resources and a need for predictability. We also believe these assumptions are common in 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. Thus, even with the static system assumption, an implementation of the model is still applicable to many systems of interest. We leave implementation of extensions that facilitate the dynamic creation and binding of reactive components for future work.

Compilation. To facilitate a design and development process based on composition and decomposition, we require a high-level language that resembles the model and examples presented in section 3. The goal of the compiler, then, is to

translate the high-level language to the language of the α-machine. In addition to conventional semantic analysis, e.g., type checking, the compiler will check that the reactive system is well-defined and provide a concurrency analysis to be used during scheduling.

Substitutional equivalence provides the logical foundation for testing well-definedness. A system is represented as a top-level reactive component. The input and output ports original to the top-level component can be converted to internal ports since the top-level component does not need an external interface. Substitutional equivalence allows the declarations of member components to be replaced with their definitions according to the procedure outlined in section 3. This process can be repeated until all member components are "inlined" resulting in a top-level component that only contains state variables, internal ports, α-statements, β-statements, and γ-statements. At this point, all internal ports, γ-statements, and β-statements can be eliminated by substituting definitions. The resulting top-level component only contains state variables and α-statements. To be a faithful implementation of the model, each α-statement must be a deterministic state transformation. The main challenge, then, lies in the co-design of a transformational language and checks for deterministic assignment.

At a high level, we desire to treat the transformation specified by an α-statement as a function that maps a value representing the current state to a value representing the next state because it leads to a simple decidable check. An α-statement then consists of five parts: a precondition list, a precondition function, an assignment list, an argument list, and an effect function. The precondition list and argument list are lists of readable objects that form the arguments of the precondition function and effect function, respectively. The assignment list is a list of writable objects that are assigned the values produced by the effect function (state variables that are not named in the assignment list retain their previous values and ports that are not named in the assignment list are undefined). There is no notion of state or assignment inside the effect function, which thus resembles a pure functional or applicative program. A conservative but decidable check, then, is to ensure that a writable object appears at most once in the assignment list.

The approach outlined above hinges on the semantics of writable objects. We define a writable object to be the name for an implied set of storage locations. All elements in the set of storage locations for a scalar variable are updated in an assignment to the variable. In contrast, only a subset of the set of storage locations for aggregate variables such as arrays and records may be updated in an assignment. Variables can also name linked data structures where the set of locations is the transitive closure of a points-to analysis [32].

For well-definedness, we require that the implied sets of storage locations for all writable objects be disjoint. Thus, an assignment to two different variables in an α-statement cannot update the same memory location. The challenge then is to show that this condition holds initially and that it holds after the execution of each α-statement. This is the subject of pointer analysis [32] which is undecidable in general. Components that communicate by passing data by

reference as opposed to by value obviously violate this condition since multiple components may then reference the same memory locations. Thus, data passed by reference through ports must have semantics that prevent concurrent updates, e.g., constant (read-only), copy-on-write, etc. The goal then is to place appropriate restrictions on the language so as to make pointer analysis feasible and enable the safe sharing of data across ports.

To achieve the general approach outlined above, an approach to persistent data structures [22] and copy elimination [27] is needed. A persistent data structure is one in which updates and queries can be made to any version, e.g., lists in LISP, Scheme, Haskell, etc. Persistent data structures match the semantics of existing functional programming languages since they allow variables to be treated as immutable values. In contrast, an ephemeral data structure is one in which updates and queries can only be to the most recent version. Ephemeral data structures are common in imperative languages and typically have simpler designs [42]. Copy elimination attempts to replace copying with in-place modification when it can be shown that the old value of a variable is no longer needed. Copy elimination is often used for large aggregate data structures, e.g., arrays, that resist efficient persistent implementations.

An approach that to our knowledge has not been attempted is to restrict a functional language solely to ephemeral data structures. We define an *ephemeral function* to be a function with an implementation (i.e., *schedule of operations*) that allows all data structures to be treated as ephemeral. This approach has the potential to be more efficient since variables may be updated without copying. Functions with no ephemeral schedule must explicitly introduce copying, which is beneficial since it makes the developer aware of such potentially costly operations. Ephemeral functions may *compose*, meaning that ephemerality may be established solely from the definition of a function and the signatures of any called functions. The challenge then is to develop an analysis to determine if a function is ephemeral and design the language of α-statement transformations based on this analysis.

The goal of concurrency analysis is to determine which α-statements are independent and therefore can be executed concurrently. Using the formulation above, we associate with each α-statement a set of objects given by forming the union of the precondition list, assignment list, and argument list (called the α-statement's *implied objects*). Two α-statements are independent if their sets of implied objects are disjoint.

α-machine. We propose a virtual machine, called an α-machine, capable of executing systems expressed as a single top-level component containing only state variables and α-statements. An α-machine has five parts:

1. The *data section* contains all of the statically allocated state variables.

2. The *heap* contains dynamically allocated state variables.

3. The *program* contains the code necessary to implement the α-statements.

4. The *stack* contains dynamic storage for function calls and expression evaluation.

5. The *scheduler* selects and executes α-statements.

For simplicity, we will limit the design to a stack-based machine since code generation for stack-based machines is straightforward. Our approach to dynamic allocation, garbage collection, etc. will be shaped by the transformational language used to encode α-statements.

Of primary interest to this research is the scheduler. The scheduler selects α-statements and executes them based on their preconditions. Furthermore, the scheduler must be fair meaning that it cannot indefinitely postpone the selection of an α-statement. One way of measuring the efficiency of a scheduler is to compare the number of preconditions that evaluate to true to the total number of evaluated preconditions. Similarly, one could measure the time spent evaluating and reevaluating preconditions to the time spent evaluating effects. Schedulers can also be compared using other criteria such as responsiveness, fairness, cache-awareness, context-switches, etc. Independent α-statements can be executed concurrently prompting the development of a concurrent α-machine and concurrent schedulers. A concurrent scheduler should take advantage of the static structure latent in reactive components and the dynamic behavior of the system being executed when making scheduling decisions.

Summary. In this section, we have proposed to implement the model described in section 3 for systems with static topologies. We proposed to develop a virtual machine called an α-machine that executes a reactive component consisting solely of state variables and α-statements. The α-machine will be a concurrent α-machine. We also proposed to develop a high-level language for specifying reactive components and a compiler that translates the specifications into an α-machine program. The main challenge is the design of the high-level language, which must be subject to an analysis for deterministic assignment.

Chapter 5

Advanced

Chapter 6

Evaluation

In this section, we propose a set of evaluations towards gauging the fitness of the model in section 3 and implementation in section 4 for designing and implementing real reactive systems. The first evaluation applies the model of reactive components to the design and implementation of an embedded web server. The goal is to understand the strengths and weakness of the model as it is applied to different problems in reactive systems and whether the model does indeed support principled composition and decomposition. The various components of the embedded web server will be compared to their analogues developed using different approaches, i.e., threads and events, to gain insight into the model. The second evaluation will focus on a set of micro-benchmarks for measuring the performance of the implementation to determine if the resulting overhead of using the model is reasonable. From the perspective of a software engineer, the first part of the evaluation provides guidance as to what to expect when using the model while the second part of the evaluation establishes the cost, i.e., overhead, of using the model.

6.0.1 Qualitative Evaluation

To determine whether (and if so how effectively) the model of section 3 and implementation of section 4 can facilitate the design and development of real reactive systems using principled composition and decomposition, we focus on building a real (if small-scale) reactive system using reactive components. Specifically, we will examine an "embedded" web server that is constructed without the benefit or hindrance of an operating system. Web servers have the advantage that they are both well-understood and relevant to the existing state of the art. Focusing on a well-understood problem allows us to treat the problem as a "control" while treating the model as the "test" resulting in an implementation with its own characteristics. While the proposed web server will not approach the sophistication of an industrial grade server, it will be of sufficient complexity to be representative of real-world problems. Web servers also have some "obvious" parallelism since requests are largely independent. An embedded web server re-

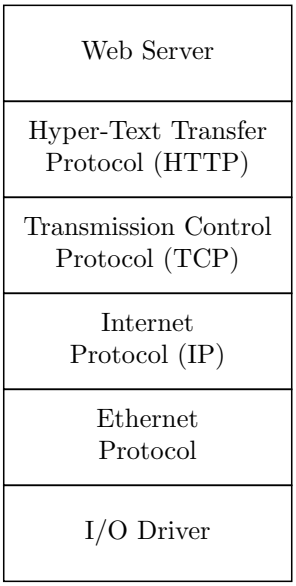

Figure 6.1: The architecture of an embedded web server

quires the development of a network protocol stack starting at the driver level. This provides an opportunity to touch on a number of areas in reactive systems such as hardware, networking, and remote interactivity.

Figure 6.1 shows one possible architecture for an embedded web server. We argue that an architecture like the one shown in figure 6.1 is the beginning of a design based on principled decomposition. Each layer can be implemented by a reactive component that interacts with the component immediately above and below it and one can envision requests propagating upward while responses propagate downward.

The lowest layer, indicated by "I/O Driver", represents the primitive I/O mechanisms available on a given architecture. On the x86 architecture, for example, this includes memory mapped I/O, port I/O, and interrupts. Using the primitive I/O mechanisms, one can construct a driver for an Ethernet network interface card. The development of device drivers is common to reactive systems such as embedded systems and operating systems and the corresponding development of a device driver will test the model's suitability for abstracting and interacting with hardware.

As with existing Ethernet drivers, e.g., network device drivers in Linux [17], the Ethernet driver will be built around two queues for sending and receiving Ethernet frames, respectively. However, we expect to see two major differences with the reactive component approach. First, we expect to see an elimination (or reduction) is the amount of synchronization code in the driver. Device drivers in Linux must be re-entrant since the Linux kernel is multi-threaded [17]. The atomicity guarantees of the model for reactive components should obviate

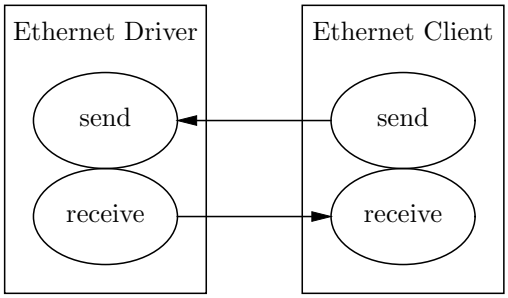

Figure 6.2: A block diagram of the Ethernet driver and a generic client

the need for most if not all of the synchronization code. Second, we expect a more direct interface to the Ethernet driver. Device drivers in Linux send and receive events using function callbacks and function calls [17]. A driver based on reactive components can use ports to express the same idea directly, i.e., the driver contains an input port for accepting frames to be sent and an output port for producing frames that have been received. Figure 6.2 shows the relationship between the Ethernet driver and a generic Ethernet client.

As is commonly done in practice, Ethernet can be used to implement the internet protocol (IP) and the transmission control protocol (TCP). These protocols are well-understood, well-documented in precise standards, and a number of reference implementations exist, thus, the development of the corresponding components is an exercise in translating existing specifications into component implementations. The experience gained by developing network protocols like IP and TCP will indicate if the model for reactive components is a good candidate for implementing other protocols. Specifically, we expect concise and direct implementations since many protocols are specified using automata, e.g., the state transition diagram of TCP. The congestion control algorithm of TCP also presents an interesting opportunity as we expect that different congestion control algorithms can be encode as components resulting in a pluggable architecture.

As is commonly done in practice, TCP/IP can then be used to construct a server capable of speaking the hyper-text transfer protocol (HTTP). The HTTP server represents the generic part of the HTTP protocol, e.g., encoding/decoding messages, while the "Web Server" layer of figure 6.1 represents the application providing the actual content and service. Using the embedded web server, we propose to implement a multi-user web-based chat service. Users may either join an existing public chat room, create a new public chat room, create a private chat with an acquaintance, invite an acquaintance to a public or private chat room, or accept an invitation to join a chat room.

The HTTP server and web server of figure 6.1 are at a similar level of abstraction as many existing reactive applications since the lower layers, i.e., TCP/IP, Ethernet, etc., are typically part of the operating system. Thus, we can compare the design of the HTTP server and web server to existing systems, e.g., the

Apache web server [24], and existing design techniques, e.g., [49]. Specifically, we are interested in how various design patterns translate into the model of reactive components and the effect on the resulting architecture. For example, we believe the model for reactive components obviate the need for the Reactor design pattern [49] due to an absence of blocking I/O operations. Similarly, we can consider patterns that may not be possible such as the Half-Sync/Half-Async design pattern [49] which reintroduces blocking operations after they have been removed by the Reactor design pattern. For each design, we will attempt to identify the design patterns that were applied, the motivation behind the use of a design pattern, and corresponding liabilities that accompany the use of a design pattern. Furthermore, we would like to provide some idea as to the influence of the model and programming the language on the implementation of the pattern, i.e., does the model or language help or hinder the expression of a pattern.

6.0.2 Quantitative Evaluation

The goal of the quantitative evaluation is to quantify the performance of the systems that constitute the implementation of section 4. In this regard, there are two systems of interest. First, the compiler will rely on a number of new analyses. Thus, we will derive complexity bounds for theses analyses but also measure the real performance of these algorithms to determine if they are feasible in practice. Second, the performance of a system expressed as an α-machine image is linked to the performance of the scheduler. Thus, we will use static analysis and micro-benchmarks to measure the performance of the scheduler.

With respect to schedulers, we are interested in three quantities corresponding to the efficiency of the scheduler, the time overhead of scheduling, and the storage overhead of scheduling. To illustrate, consider a brute-force scheduler that merely cycles through the list of α-statements. Such a scheduler is obviously fair but may not be efficient. That is, the scheduler may evaluate a number of preconditions before finding an α-statement that is enabled. Thus, one way of measuring the efficiency of a scheduler is to compare the number preconditions that evaluate to true to the total number of evaluations. Similarly, one could instead measure the time used to evaluate preconditions, i.e., a scheduler should avoid evaluating complex preconditions when possible. The time overhead or computational complexity of scheduling is the time required to select the next α-statement. The complexity of the brute-force scheduler is $O(1)$ since the next statement to be selected is known based on the current statement. Finally, improving the efficiency or time overhead of scheduling may require additional data structures. The storage overhead of scheduling, then, is any overhead associated with state variables, α-statements, etc.

For concurrent α-machines, we must also evaluate overheads associated with synchronization. Most likely, concurrent α-machines will be implemented using threads as this is the native concurrency mechanism available on existing processors. A concurrent scheduler must use synchronization primitives to coordinate the activities of the threads. The time spent on synchronization is

necessary but reduces the overall performance of the scheduler. Single-threaded and multi-threaded schedulers may be able to take advantage of static analysis, e.g., statically partition and schedule α-statements based on data flow, and/or dynamic analysis, i.e., use the dynamic behavior of the system to improve scheduling decisions.

Chapter 7

Conclusion

A reactive system is characterized by "ongoing interactions with its environment[40]." Asynchrony and concurrency are two features of reactive systems that make them inherently difficult to develop. Reactive systems are already used in 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 we would like to use when designing and implementing reactive systems given such increases in complexity. We argue that the techniques based on explicit atomicity and deterministic sequencing prevent decomposition and composition and contribute to the accidental complexity associated with reactive system development.

Broader impact. 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 and continue to amaze 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.

Contributions. In this work, we propose three contributions to the state of the art in reactive systems development. First, we propose a new model called reactive components for reactive systems based on direct support for reactive semantics and principled composition and decomposition. A reactive component is a set of state variables and non-deterministically scheduled atomic transitions. Transitions in different components can be linked via ports which also allow data to be exchanged among components. Implicit atomicity and non-deterministic

sequencing allow reactive systems to be designed and implemented through principled decomposition and composition.

Second, we propose to implement the model of reactive components to determine whether the assumptions and semantics of the model can be realized using existing techniques and architectures. For tractability, we limit the implementation to systems with a fixed configuration of reactive components. The major challenge when implementing the model is to enforce the semantics that require all state transitions to be deterministic. Our approach is based on the encoding of transitions using a pure functional or applicative language which in turn requires an approach to data structures (persistent vs. ephemeral). The platform will consist of a compiler for a high-level language shaped by the semantics of reactive components and a virtual machine.

Third, we propose to evaluate the model of reactive components and its implementation. In the first part of the evaluation, we will apply the model to the design and development of an embedded web server. The goal of the evaluation is to gauge the fitness of the model and usefulness of principled composition and decomposition by applying the model to a representative real-world reactive system. Developing an embedded web server allows the model to be applied to a number of problems in reactive systems such as hardware, network protocols, and interactive applications. The second part of the evaluation is a quantitative measurement of the implementation (compiler and virtual machine) to ensure that the employed algorithms and techniques result in a practical engineering tool.

Timeline. We propose the following timeline for the different contributions: 4 months for the model (section 3), 12 months for the implementation (section 4), and 5 months for the evaluation (section 6).

Beyond this dissertation. While we argue that the static system assumption is valid, it is nevertheless restrictive and prevents designs that may be more naturally formulated using a unbound number of components. Thus, we believe the first step to applying reactive components to a broader range of systems is an extension to dynamic systems. Another barrier to the wide-spread adoption of reactive systems is its reliance on a virtual machine. As indicated by the Java programming language, a virtual machine is appropriate for application level programming while lower-level programming is often performed in a compiled language such as C. Thus, the compiler proposed in section 4 may serve as the beginning for a compiler capable of emitting machine code. The main direction for future work is to continue to gain experience with reactive components by implementing systems. The class of reactive systems is enormous and spans everything from the programs that control 8-bit micro-controllers to the distributed applications running Google's cluster. We believe the evaluation of section 6 is an appropriate first case study but necessarily lacks the depth and breadth needed to fully evaluate reactive components.

Outcomes. The proposed research will be successful if it generates new knowledge regarding the design and implementation of reactive systems. The conclusion of this research will be a explanation as to whether or not reactive components are a viable approach to reactive systems and if so, what additional research is warranted. If the research concludes that reactive components are not a viable approach, it will reveal the characteristics of reactive components and more generally models based on the non-deterministic sequencing and implicit atomicity that make them an inappropriate or impractical foundation for reactive systems. If reactive components are indeed a viable approach to reactive systems, then the proposed work could represent the beginning of a significant shift in the theory and practice concerning reactive systems. That is, reactive components have the potential to replace threads and events which are the dominant approaches to reactive system development today.

Bibliography

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

[2] *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.

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

[4] 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.

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

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

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

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

[9] 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.

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

[11] 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.

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

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

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

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

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

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

[18] 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.

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

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

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

[22] 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.

[23] 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.

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

[25] 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.

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

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

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

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

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

[36] 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.

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

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

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

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

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

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

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

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

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

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

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

[48] 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.

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

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

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

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

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

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

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