International Journal of Cultural Inheritance & Social Sciences (IJCISS)
Vol. 6 Issue 11, March 2024, ISSN: 2632-7597 Impact Factor: 5.266
Journal Homepage: http://ijciss.com/, Email: journal.ijciss@gmail.com
Double-Blind, Peer Reviewed, Refereed & Open Access International Journal

A TRANSACTIONAL MODEL AND PLATFORM FOR DESIGNINGAND IMPLEMENTING REACTIVE SYSTEMS

Esam Mohamed Elwan
Professor of Strategic management and computer management
Head of Information Technology and Human Development Consultant
Al Ain University of Science and Technology, U. A. E.
Email: dresamelwan@gmail.com

Abstract

A reactive program is one that has “ongoing interactions with its environment”.
Reac- tive programs include those for embedded systems, operating systems, network clients and servers, databases, and smart phone apps. Reactive programs are already a core part of our computational and physical infrastructure and will continue to proliferate within our society as new form factors, e.g. wireless sensors, and inexpensive (wireless) networking are applied to new problems.
Asynchronous concurrency is a fundamental characteristic of reactive systems that makes them difficult to develop. Threads are commonly used for implementing reactive systems, but they may magnify problems associated with asynchronous concurrency, as there is a gap between the semantics of thread-based computation and the semantics of reactive systems: reactive software developed with threads often has subtle timing bugs and tends to be brittle and non-reusable as a holistic understanding of the software becomes necessary to avoid concurrency hazards such as data races, deadlock, and livelock, Based on these problems.

Keywords: reactive program, ongoing interactions, embedded systems, operating systems, databases, smart phone apps, Interactive programs, systems, the educational field.

Introduction:

1. 1 Reactive Programs and Systems

Manna and Pnueli classify programs as either being transformational or reactive . As implied by their name, transformational programs transform a finite input sequence into a finite output sequence. Transformational programs often can be divided into

three distinct phases corresponding to the activities of input, processing, and output. A compiler is an example of a transformational program as it transforms a source file into an object file. Formal models of computation like the Turing machine and λ-calculus are concerned with transformational programs.

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

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

Asynchronous concurrency is a fundamental characteristic of reactive systems that makes them difficult to design and develop. Concurrency refers to the idea that a reactive program and its environment may act at the same time. In synchronous models, a reactive program and its environment share a common clock that allows the reactive program to coordinate access to shared resources, e.g., the environment writes a shared variable in one clock cycle and the reactive program reads it in the next. However, shared clocks are difficult to im- plement and do not scale. Consequently, many reactive systems are asynchronous, meaning that the reactive program and the environment evolve independently with respect to time. To facilitate communication, asynchronous models include facilities for atomicity that allow a reactive program to act

without being interrupted by its environment and vice versa. These facilities for atomicity allow a reactive program to synchronize with its environment, as the atomic events play the role of a shared clock. A common approach to ensuring correctness in asynchronous models is to view the environment as an adversary that may deliver inputsat inopportune times.

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

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

Many reactive programs are multi-threaded and are designed to use the facilities of a con- ventional operating system. For the discussion to follow, we consider each thread in a multi-threaded program to be a reactive program. The environment for a thread then is the operating system and the other threads with which it directly interacts. This

matches the definition of a reactive program as each thread will have ongoing interactions with its envi- ronment, i.e., interact with the operating system and other threads. This also matches the definition of asynchronous concurrency as the thread and its environment share no common clock and can interact asynchronously. To be more specific, a system call is asynchronous from the perspective of the operating system and threads may receive asynchronous signals from the operating system and from each other.

1.2: Trends

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

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

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

1.3 Limitations of the State of the Art

Failure to account for asynchronous concurrency may result in a reactive program with timing bugs, meaning that correct execution depends on arbitrary scheduling decisions. Timing bugs may be manifested when the schedule of events is perturbed, e.g., due to the introduction of a new processor or operating system, or when part of the program takes more or less time than normal. Operating systems are particularly susceptible to timing bugs caused by device drivers.

Timing bugs can escape even a rigorous software development process and may lay dormant for years. 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) . Timing bugs can cost many hours of debugging time and can lead to a poor user experience, e.g., a non-responsive device or application, and loss of revenue, e.g., when an advertisement service cannot be used while a server reboots.

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

A number of design patterns for concurrency have been developed to help developers avoid concurrency hazards,. These design patterns represent a move toward implicit atomicity as they often attempt to leverage language features to control atomicity (e.g.,

scoped locking). While some of these patterns have been incorporated into programming languages (e.g., the synchronized keyword of Java implements the thread-safe interface pattern), they are most often enforced only by convention and therefore easily violated or ignored (e.g., if a new developer is unaware of the convention). In practice, explicit atomicity and the corresponding use of synchronization primitives has proven to be tediousand error prone.

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

1.4 Challenges

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

Reduce accidental complexity. A key challenge towards adequately supporting complex reactive systems is to reduce the accidental complexity associated with their design and implementation. For software, accidental complexity is defined as the "difficulties that today attend its production but that are not inherent ." 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 . As described in the previous section, we claim that the currently dominant approaches to developing reactive systems rely on inherently transformational languages that have been

augmented with features for concurrency, which introduces an example of the kind of problem that Lee has identified. Thus, reducing the accidental complexity in reactive systems requires an approach that provides direct support for reactive semantics and addresses the inherent difficulties of asynchronous concurrency.

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

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

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

1- The model should define units of composition and a means of composition.

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

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

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

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

5- Composition should be *compositional* meaning that the properties of a unit of compo- sition 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 sys- tem X that is a pipeline formed by composing a filter system F with reliable FIFO channel system C. This principle states that the properties of the composed Filter- Channel X can be expressed in terms of the properties of the Filter F and Channel C. Compositionality requires the ability to establish properties for units of composition that cannot be violated through subsequent composition.

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

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

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

1.5 Approach and Contributions

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

independent of control flow. This relieves developers from needing a holistic understanding of the call graph when composinga complex state transition.

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

** Background and Related Work:

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

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

Computation in a reactive system then can be viewed as a sequence of state transitions .

As these transitions may be complex, platforms typically allow complex state transitions to be composed from primitive state transitions and complex states, e.g., arrays, records, tuples, lists, sets, etc., to be composed from primitive states. Three orthogonal techniques for composing complex state transitions are expressions, sequential composition, and paral- lel 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 de- composed 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 . Conceptually, the right-hand side (RHS) of a parallel assignment is computed before any of the variables on the left-hand side (LHS) are modified.

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

Multiple reactive processes effect state transitions that overlap in time, which is called *con- currency*. 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 over- lapped by interleaving the primitive transitions of the corresponding complex transitions. The result of concurrent state transitions that update the same (shared) state may be un- defined. 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 withanother 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.,.

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 im- plied 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 and Hoare's Communicating Sequential Processes.

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 *sched- uler*, 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 and Lynch's I/O Automata.

Atomicity may either be *explicit* or *implicit* in a model for reactive systems. Platforms that support reactive processes with deterministic sequencing and shared

variables typically include primitive transitions called *synchronization primitives*, e.g., test-and-set, compare- and-swap, that may atomically update state and/or alter the flow of control. Synchro- nization primitives can be used to construct more general synchronization mechanisms like semaphores and monitors. The goal of synchronization is to create atomic sequences of transitions called *critical regions* or *critical sections*. Atomicity, therefore, is made explicit by the programmer. Message passing combines communication with synchroniza- tion to achieve implicit atomicity in reactive programs based on deterministic sequencing . 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 Coop- erating Sequential Processes. Andrews and Schneider describe a number of techniques associated with this approach including coroutines, fork/join, spin locks, semaphores, con- ditional critical regions, and monitors. This model is supported by widely available platforms, i.e., operating systems, via processes with shared memory or threads.

A number of architectural patterns have been developed based on this approach, e.g.,.

As described by Lee , 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 , Split-C , C++11 . Sutter and Larus and Lee provide a modern perspective on the difficulties associated with this approach. In , the authors call for "OO for concurrency–higher-level abstractions that help build concurrent programs, just as object-oriented abstractions help build large componentized programs." The work presented in this dissertation is a step in this direction.

Transactional memory has been proposed as an alternative to locks when synchronizing mul- tiple processes. Transactional memory was inspired by the atomic transactions of databases . Knight and then Herlihy and Moss proposed cache-based hardware support for trans- actional memory. Transactional memory is forthcoming on modern processors .

Software transactional memory, proposed by Shavit and Touitou , has sparked a great deal of in- terest and has been implemented in a number of languages, e.g., Clojure .

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

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

This model is also supported by widely available platforms via pipes, message queues, and sockets.

Go is a modern programming language with language support for the Communicating Sequen- tial Processes model . A message passing channel may either be unbounded or have a fixed size and may be accessed synchronously or asynchronously by either the sender or thereceiver .

Events, as proposed by Ousterhout , and embodied in the Reactor and Proactor architectural patterns [84], offer a popular technique for structuring user applications based on deterministic sequencing. The application is designed around a loop that multiplexes I/O events from the operating system using a polling function like select or poll. In response to an I/O event, the application invokes an (atomic) event handler that may update the state of the process and perform non-blocking I/O on various channels. Events invert the flow of control since high-level functions, e.g., processing a message, are triggered by low- level functions, e.g., receiving a byte. The context of each computation must be managed explicitly which is referred to as "stack ripping".

Event handlers from different logical computations may be interleaved giving the illusion of concurrency while avoiding the chal- lenges of synchronization. Event systems wishing to take advantage of true concurrency must use multiple event loops and face all of the challenges of multi-threaded programming. Node.js and ECMA Script (JavaScript) are two modern programming languages based on an event loop.

UNITY and I/O Automata. The UNITY model of Chandy and Misra and the I/O Automata model of Lynch are two influential models for reactive systems based on non-deterministic sequencing. In these models, a scheduler repeatedly executes transitions selected non-deterministically from the set of possible transitions. The

scheduler is assumed to be fair, meaning that it will execute a transition an infinite number of times in an infinite execution. Transitions correspond to *(parallel) assignment statements* in the UNITY model and *actions* in the I/O Automata model. The UNITY model contains two means of com- posing programs which suggests that a model based on non-deterministic sequencing could support principled decomposition. Creating a program via the *union* operation involves taking the union of the state variables (name-based equivalence) and assignment statements of the constituent programs. *Superposition* is a means of composition that transforms an underlying program into another. The transformation is allowed to add new state variables, add new assignment statements with the limitation that they only update new variables, and augment existing assignment statements but only by adding clauses that modify new variables. The size of the resulting program (measured in assignment statements) is on the order of the sum (as opposed to the product) of the sizes of the two constituent programs.

Superposition is property-preserving while union is not property-preserving. However, su- perposition has certain weaknesses, as described by the authors of UNITY.

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

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

The essence of superposition is that a transition in one program can be linked to a transition in another program, i.e., parallel composition. Whereas UNITY lacks language support for doing so, the I/O Automata model presents a partial solution by associating names with transitions (actions). Actions in different Automata can then be composed on the basis of name matching. The I/O Automata model defines three kinds of actions: output actions, input actions, and internal actions. Output actions and

internal actions, collectively called local actions, contain guards and may be executed by the scheduler. Each input action must be composed with an output action to be executed. Output actions can produce values that are consumed by input actions. Since the state of each I/O automaton is private, composition in the I/O Automata model is property-preserving.

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

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

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

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

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

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

***** Reactive Component Model:**

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

3. 1 Features of the Model:

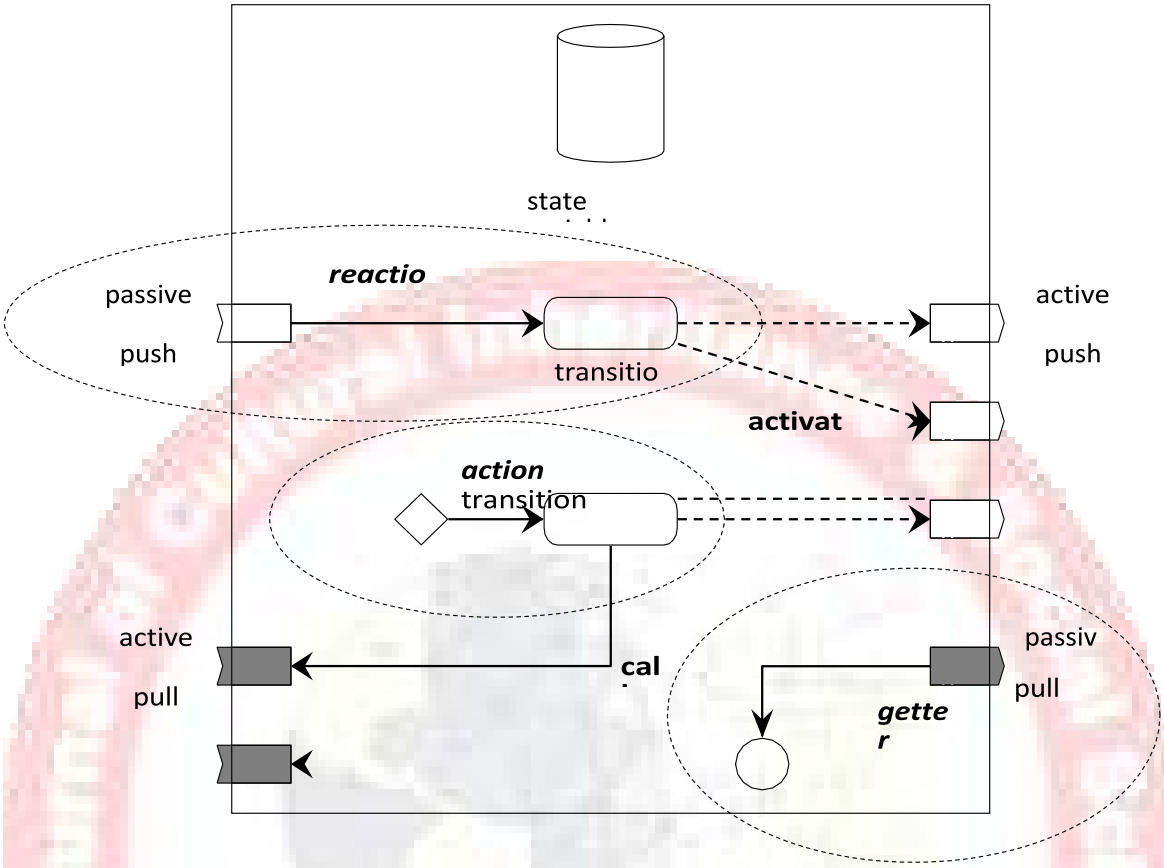

Figure 3.1: Features of a reactive component

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

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

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

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

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

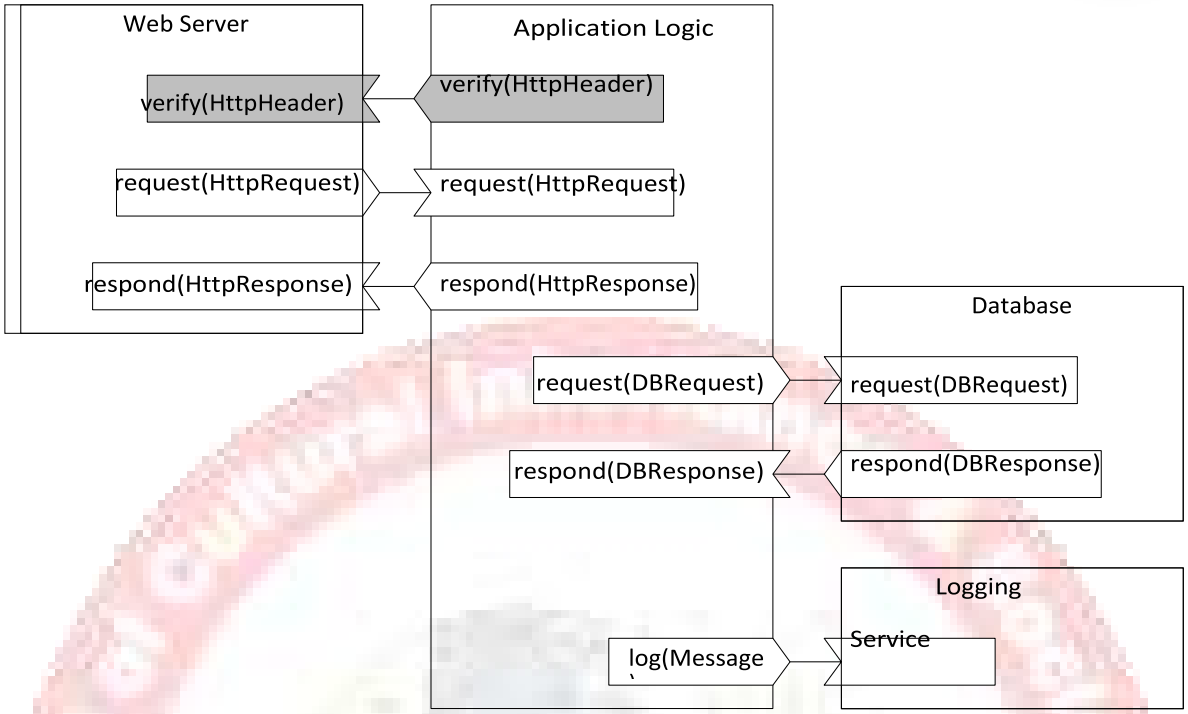

Figure 3.2: Diagram of a web application built using reactive components

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

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

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

3. 1. 1 State Variables:

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

3.2.1 Atomic State Transitions:

The rest of the internal core of a reactive component is a set of atomic state transitions that manipulate the state variables. A precise definition of the atomicity of state transitions will be deferred to the subsequent sections on composition (Section 3.1.3) and execution (Section 3.1.5). State variables and state transitions are private, meaning that transitions can only refer to the state variables of the associated reactive component and a transition in one reactive component can only be linked to a transition in another component through the composition mechanisms. Thus, all initialization and updates to state variables can be determined by examining the transitions of the corresponding component. The encapsulation of state variables contribute to composition being *compositional*. All properties established from the transitions of a component will continue to hold as the component participates in composition since the set of transitions that affect the state

variables is fixed. State transitions must be deterministic meaning that the next value of every state variable is uniquely and well defined. State transitions defined by a sequence of simple assignments are implicitly deterministic. However, special care must be take to ensure that state transitions are deterministic when described using parallel assignment statements as the same variable may appear on the left-hand-side and be assigned two different values . As described in Sections 3.1.3 and 3.1.5, composition links transitions using parallel assignment which creates an opportunity for non-deterministic state transitions. The problem of non-deterministic state transitions arising from composition is explored in Section 3.3.

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

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

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

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

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

3.1.4 Transactions:

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

3.1.5 Execution:

We adopt the common practice of modeling concurrency with non-deterministically executed atomic actions as is done in UNITY , I/O Automata , and the Actor Model . The execution of transactions is performed by a scheduler. The scheduler executes one transaction at a time, thus, each transaction is atomic with respect to all other transactions. A transaction is enabled if its precondition is true. When executing a transaction, the scheduler first evaluates the precondition and then executes the body of the transaction if the transaction is enabled. Thus, executing an enabled transaction

may result in a state change while executing a disabled transaction never causes a state change. The scheduler is fair meaning that each transaction is executed an infinite number of times. The system may reach a fixed point where all transactions are disabled. The order in which transactions are executed is not determined, thus, execution is non-deterministic.

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

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

2 - AIRIAM RM and AIRIRM AM. These sequences resolve the issue with the first se- quence by providing a clear immutable phase (AIRI) and mutable phase (AM RM and RM AM).

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

3.2: Example: Clock System:

To illustrate the behavior of reactive components under this model, we rewrite the

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

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

```
component Clock {
var int counter (0)
var bool flag (false)
push clock(int t)
reaction request() flag: = true
Clock: flag -> flag: = false activates clock(counter)
Tick: counter: = counter + 1
}
```

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

```
component Client {
var bool flag (false) push request()
Request: !flag -> flag:= true activates request()
reaction clock(int t) flag: = false || /* do something with t */
  }
```

Figure 3.4: Definition of the Client of the Clock System

```
component System{
var Clock clock
var Client client
bind {
client.request ->
clock.requestclock.clock ->
client.clock
   }
}
```

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

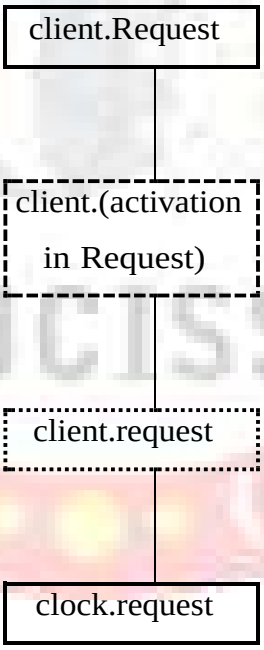

Figure 3.6: Transaction diagram for the client.Request action of the Clock System

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

3.3 Properties of Composition:

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

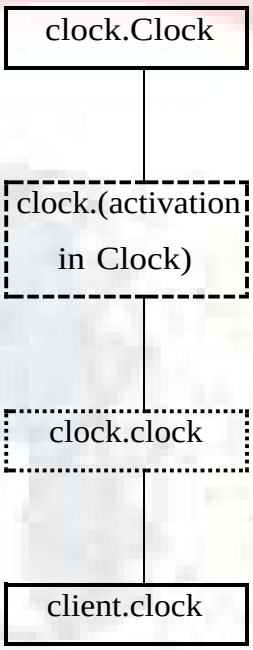

Figure 3.7: Transaction diagram for the clock.Clock action of the Clock System

component System {

/* Substitution of clock component. */ var int clock_counter (0)

var bool clock_flag (false) push clock_clock(int t)

reaction clock_request() clock_flag:= true

clock_Clock: clock_flag -> clock_flag:= false activates

clock_clock(clock_counter) clock_Tick: clock_counter:= clock_counter + 1

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

push client_request()

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

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

client_request -> clock_request clock_clock -> client_clock

}

```
}
```

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

```
component System {
var int clock_counter (0) var bool clock_flag (false) var bool client_flag (false)
push clock_clock(int t)
push client_request()
clock_Clock: clock_flag -> clock_flag, client_flag:= false, false activates
clock_clock(clock_counter) ||
/* do something with clock_counter */
client_Request: !client_flag -> client_flag, clock_flag:= true, true activates
client_request()
clock_Tick: clock_counter := clock_counter + 1
}
```

Figure 3.9: Simplifications of ports, bindings, and transitions in the expanded System com- ponent of the Clock System

3.3.1 Substitutional Equivalence:

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

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

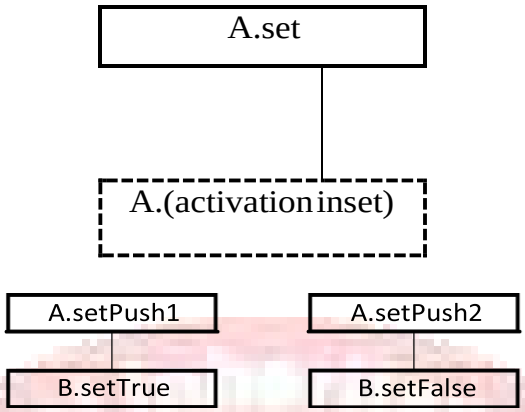

Figure 3.10: Transaction diagram for a non-deterministic transaction

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

3.3.2 Determinism and Composition:

The result of composing two well-defined reactive components may yield a system that is non- deterministic. The two "hazards" that must be avoided are non-deterministic assignment to state variables and recursively activated reactions. Non-deterministic assignment results when the next value for a state variable is not well defined due to the inclusion of two or more transitions in a transaction that operate on the state variable. To illustrate, consider the transaction depicted in Figure 3.10. The set action of component A has a single activation that activates both the setPush1 and setPush2 push ports. The setPush1 port is bound to the setTrue reaction of component B while the setPush2 port is bound to the setFalse reaction of the same component B. Suppose that the setTrue reaction sets a flag to true while the setFalse reaction sets the same flag to false. The transaction is non-deterministic because the value of the flag after the (A,set) transaction may either be true or false. An offending pair of transitions may appear anywhere in a transaction graph given that arbitrary transaction graphs may be constructed through composition. If the underlying language used to define transitions admits pointers, dynamic memory, if statements, and loops (i.e., is Turing-complete), then the problem of determining if two transitions operate on the same state variable is undecidable in general [59, 80]. For a transaction graph G, instances i1and i_2, and actions/reactions t_1 and t_2, a necessary (but not sufficient)

condition for non- deterministic assignment *NDA* from composition is *NDA*(*G*): $\exists(i_1, t_1), (i_2, t_2) \in G.N, i_1 = i_2, t_1 /= t_2$ which says that the transaction must contain two different transitions involvingthe same component instance.

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

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

3.3.3 Decomposition, Getters, and Pull Ports:

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

Another motivation for decomposition is potentially increased performance through paral- lelism. Recall that true concurrency in reactive components is modeled as the serial and non-deterministic execution of atomic actions. Two transactions can be safely executed

con- currently if it can be shown that the state variables involved in each transaction are disjoint. As was previously mentioned, making this determination is undecidable for Turing-complete languages in the general case. However, the problem becomes decidable if the component instance is used as a proxy for its constituent state variables. Let *rw:* $t \rightarrow$ {*Read, Write*} be a function that maps a transition to a value indicating that variables are read-only during the transition or written in some way. Two instance/transition pairs are independent (*indp*) if either the instances are different or at most one of the transitions writes to the variables of the instance:

$$Indp\ ((i_1, t_1), (i_2, t_2)):\ i_1\, i_2 \vee \neg(rw(t_1) = Write \wedge rw(t_2) = Write) \quad (3.1)$$

Two transaction graphs are independent if all of their nodes are independent indp(G1, G2) =∀(i1, t1) ∈ G1.N, (i2, t2) ∈ G2.N indp((i1, t1), (i2, t2)). The significance of the preceding analysis is that the determination about what actions can be executed concurrently becomes machine checkable due to the strong guarantee that state variables belonging to an instance can only be modified by the transitions of that instance.

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

component Counter { var int counter (0)
Tick: counter:= counter + 1 getCounter() int {
return counter
}

```
}
```

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

```
component Clock { var Counter c
var bool flag (false) push clock(int t)
reaction request() flag:= true
Clock: flag -> flag:= false activates clock(c.getCounter())
}
```

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

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

The logic associated with the flag state variable represents a generic request-response pro-tocol except for the call to c.get Counter().

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

```
component RequestResponse { var bool flag (false) pull getValue() int
push response(int t)
reaction request() flag:= true
Response: flag -> flag:= false activates response(getValue())
}
```

Figure 3.13: Definition of the Request Response component of the Factored Clock System

```
component Clock { var Counter c
var Request Response rr
bind {
```

c.get Counter -> rr.get Value

}

export rr.request as request export rr.response as clock

}

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

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

3.4 Summary:

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

In this chapter, we have presented the reactive component model for reactive programs. A reactive component consists of a set of state variables and transitions that are private to the component. The interface of a reactive component consists of push ports and reactions, which allow a component to trigger a transition in another

component, and pull ports and getters, which allow a component to access the state of another component. The external or visible behavior of a reactive component can be traced through its interface, specifically its push ports. The internal details of a component can often be abstracted away to permit reasoning about the behavior of a composed system at various levels of detail. Composition is achieved through recursive encapsulation (sub-components) and explicit port binding and satisfies the requirements for principled composition set forth in Section 1.4. As demon- strated in Section 3.3.1, the definitions of sub-components can be substituted into the con- taining component resulting in an equivalent system (substitutional equivalence). Similarly, sub-components may be "factored out" by using pull ports and getters to safely access the state of the sub-components.

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

**** The rcgo Programming Language:**

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

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

4.1 Challenges:

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

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

4.2 Constraints:

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

Strict enforcement of the reactive component model. To enforce the semantics of reactive components, the checks of interest are 1) the separation of the immutable and muta- ble phase (Section 3.1.5), 2) the binding of pull ports (Section 3.3.3), and 3) the detection of non-deterministic state transitions arising from composition (Section 3.3.2). Not performing these checks is unacceptable due to the subtlety associated with

developing correct reactive programs: the semantics of reactive components would be enforced only through convention which is easily violated. Similarly, placing the burden on developers is unacceptable due to the amount of detail that must be considered. Thus, the implementation must enforce the semantics of reactive components through adequate checking.

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

Efficient inter-component communication. Linked data structures also create an opportunity for efficient communication between components. There are three modes by which a component (sender) can share information with another component (receiver) as they in- teract through push ports and pull ports. First, the sender may use *value semantics* where it provides complete copies of the values to be communicated. This approach is reasonable for values like numbers and small records. Second, a sender may use *reference semantics* where it provides a pointer (reference) to the data to be communicated. This approach is reasonable when the sender offers up a large data structure. The receiver is responsible for copying any data that needs to be retained. When using such reference semantics, receivers may read and copy the data structure represented by the pointer but may not alter or per- sistently remember the original data structure in any way. Third, a sender may use *move semantics* where it provides a pointer to a data

structure that the receiver may adopt as its own. In this case, the sender promises to "forget" the data structure as the receiver is the new owner of the data structure. Move semantics are appropriate when the size of the data to be communicated is large, there is a single receiver, and the sender does not need to retain the data. These conditions may arise, for example, in situations involving networking stacks and pipelines. Without move semantics, the practicality of the reactive component model is greatly diminished.

4.3 Approach:

Our approach to implementing the reactive component model defined in Chapter 3 is to provide a programming language that facilitates the direct expression of reactive components. Language support for the model is beneficial because it closes the semantic gap between reasoning and implementation. The importance of language support can be seen in techniques like structured programming and object-oriented programming .

While these techniques can be applied in virtually any setting, their lasting utility is derived from their implementation in a variety of programming languages. Providing language support for reactive components raises the level of abstraction and allows reasoning about a system consisting of them directly from its specification, instead of reasoning in one set of semantics while implementing in another which can be tedious and error-prone.

Language support allows developers to rely on the consistent application of the semantics of the model through strict enforcement. Designing language support specifically for reactive components creates an opportunity to introduce syntax and semantics that allow a compiler or interpreter to distinguish programs that conform to the semantics of reactive components from those that *potentially* may not. To illustrate, consider the problem of ensuring that the state of a component does not change during the immutable phase of a transition. We observe that an action/reaction is similar to a method. Based on this observation, checking the immutability of a component in the immutable phase should be similar to checking for const correctness in C++. In C++, the const correctness check is performed early in the compilation process when the program is represented as an abstract syntax tree (AST) with full semantic information, as opposed to late in the process where the program is represented by machine instructions with very limited semantic information. From this, we observe that such checking is supported by 1) adding features to the language

to provide the necessary semantic information and 2) performing the checks early in the translation process. An existing language might not provide enough detail to enable the necessary checks or to achieve them efficiently. Along these same lines, an implementation may have to make conservative assumptions to enforce the semantics of reactive components. As was seen in Section 3.3, allowing a component instance to serve as a proxy for all of its state variables allows the check for non-deterministic state transitions to be implemented using known techniques.

4.4 Preliminaries:

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

Our implementation of Go's types, declarations, statements, and expressions also is intentionally only as broad as is needed to demonstrate the contributions of this dissertation.

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

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

Static system assumption:

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

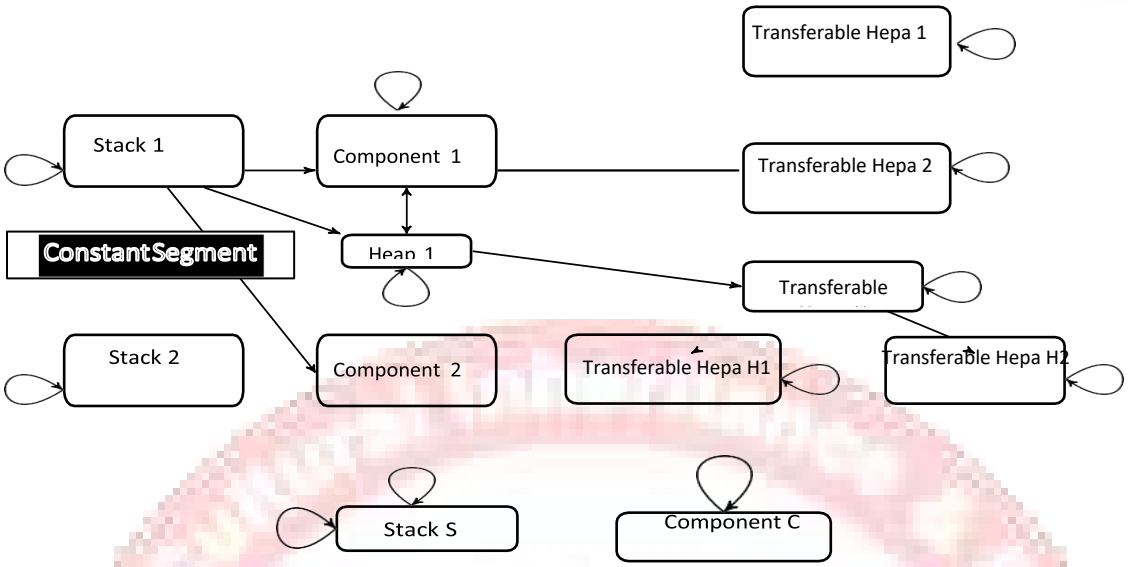

Figure 4.1: Memory model for the rc_{go} run-time system. An edge from one segment to another indicates that a pointer in one segment (source) may refer to a location in another segment (target). Any pointer may refer to a location in the constant segment (these edgesare not shown).

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

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

As shown in Figure 4.1, the other three types of segments are component segments, heap segments, and transferable heap segments. A component segment contains the statically allocated state of a component while a heap segment contains the dynamically allocated state.

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

4. 5 Syntax and Semantics:

This section describes the syntax and semantics of rcgo. Many concepts, e.g., components, ports, actions, and reactions, have a natural mapping into the Go language as types and method-like constructs. A major divergence from Go is the introduction of attributes that change the mutability of variables, as this was necessary to maintain the isolation between components while supporting reference semantics. The separation between the immutable phase and mutable phase is accomplished via an activate statement that contains a con-

tinuation to be executed in the mutable phase. To support move semantics, we introduce a new type called a transferable heap which allows one component to safely transfer a collection of objects to another component. In the following presentation, we use the following convention:

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

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

type ID component { FieldList };

For example,

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

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

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

func (ID TYPE_ID) METHOD_ID Signature Body

func (ID *TYPE_ID) METHOD_ID Signature Body

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

Intrinsic and indirection mutability. Variables and parameters are declared with both an *intrinsic mutability* and a *indirection mutability*. Intrinsic mutability limits the operations that can be performed on the lvalue of the variable or parameter while indirection mutability limits the operations that can be performed on lvalues derived from the rvalue of the variable or parameter. By default, variables and parameters have

mutable intrinsic and indirection mutability. The following code is legal and sets z to 6.

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

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

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

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

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

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

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

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

The second kind of mutability is called *foreign* mutability. Foreign mutability is like im- mutable mutability with the added condition that an address with foreign mutability cannot be stored in a component segment, heap segment, or transferable

heap segment. The pri- mary application of foreign mutability is to support reference semantics for communicationwhile enforcing the isolation of heaps.

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

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

The following checks are applied to assignment statements:

1 - The lvalue and rvalue must be type compatible.

2 - The lvalue must have mutable intrinsic mutability.

3 - If the type involved contains a pointer, check for compatible indirection mutability (see Table 4.1). Essentially, the indirection mutability of the lvalue must be at least as “weak” as the indirection mutability of the rvalue. This enforces the “stickiness” of indirection mutability.

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

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

A parameter is *foreign safe* if 1) the type of the parameter does not contain pointers or slices or 2) the parameter is declared with foreign indirection immutability. A parameter list is foreign safe if all parameters in the list are foreign safe. A signature (a parameter list and a return parameter list) is foreign safe if the parameter list and return parameter list are both foreign safe. Signatures used in inter-component communication, such as push ports

and reactions, must be foreign safe to permit reference semantics while enforcing the isolation of state between components.

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

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

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

** An initializer must have a pointer receiver to a component type.

** The signature must be foreign safe.

** An initializer may only be invoked by another initializer.

An initializer sets the heap segment on entry and resets the heap segment on exit so that all allocated memory is attributed to the receiver.

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

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

Actions. An action has the form:

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

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

Reactions. A reaction has the form:

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

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

reaction. Reactions set the heap segment on entry and reset the heap segment on exit so that all allocated memory is attributed to the receiver.

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

push (Parameter List)

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

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

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

getter (ID $const *TYPE_ID) GETTER_ID Signature Body

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

** A getter must have a pointer receiver to a component type declared with immutable indirection mutability.

** The signature must be foreign safe.

** A getter may only be invoked by an initializer, another getter, or an action or reaction in the immutable phase.

** A getter sets the heap on entry and resets the heap on exit so that all allocated memory is attributed to the receiver.

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

pull (Parameter List) Return Parameter List

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

```
type Producer component {
...
Is Out put Buffer Full pull () bool;
};
```

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

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

```
bind (ID *TYPE_ID) BIND_ID {
  Bind Statement
  ...
}
```

For example:

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

The first statement of the example binds the Out push port of the System's producer to the In reaction of the System's consumer4. The second statement of the example binds the is Out put Buffer Full pull port of the System's producer to the is Input BufferFull getter of the System's consumer. The left side of a bind statement always refers to a port while the right side refers to a getter or reaction. The direction of the arrow indicates the logical flow of information. Thus, information flows from a push port to a reaction (->) and information flows from a getter to a pull port (<-). A pull port must be bound to exactly one getter. A reaction may be bound to at most one push port. Binders are associated with a component type and evaluated for each instance of that component type.

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

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

Activate statements can only occur in the body of an action or a reaction. Assume that the receiver of the action or reaction is named this5. The expression this.PORT_ID must refer to a push port and the arguments passed to the push port must agree with its signature. The list of push ports in an activate statement is optional. When an activate statement is executed, the named push ports are activated meaning that the reactions bound to those push ports are executed with the given arguments. These reactions, in turn, may execute other activate statements. Once all of the actions and reactions in the transaction have activated their last push ports, they proceed to execute the bodies of the activate statements. Recall that the receiver this has immutable indirection mutability. Thus, all computation up to and including the last port activation constitutes the immutable phase of the transaction since the state of the components is not allowed to change. Within the scope of the body, the receiver this changes to mutable indirection mutability which then allows the state of a component to be changed. Thus, the bodies of activate statements form the mutable phase of the transaction. Parameters and variables declared with foreign indirection mutability are hidden within the body of an activate statement. This prevents one component from accessing the state of another component during the mutable phase. Actions and reactions return, i.e., their flow of control is halted, after the execution of the body of a activate statement. Activate statements guarded by if statements facilitate conditional activation. An activate statement may not appear in another activate statement. Our implementation of activate statements is described in Section 5.3.

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

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

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

```
type System component {
  clock [5]Clock;
  push [5]request ();
  ...
};
 [5] action (this $const *System) (...) {
   ...
   activate clock[IOTA] {
     ...
   };
}
```

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

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

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

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

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

The transferable heap type is so named because it resembles a heap used for dynamic

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

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

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

creates new heap with an integer root.

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

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

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

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

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

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

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

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

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

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

already been claimed.

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

4. 6 Examples

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

4. 6. 1 Shared Variable System:

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

```
type Variable component {value int;};
init (this *Variable) Init () { this.value = -1;}
[3] reaction (this $const * Variable) Set (v int) { activate {
this.value = v;}}
getter (this $const * Variable) Get () int { return this.value;}
```

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

A Variable component has a single integer state variable named value. This variable is initialized to the sentinel value -1 in the initializer Init. Notice that the initializer has a

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

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

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

type Process Status int; const PROCESS_IDLE = 0; const PROCESS_ACCESS = 1; const PROCESS_DECIDE = 2; const PROCESS_DONE = 3;

type Process component { status Process Status; input int;

output int;

get_x pull () int; set_x push (v int); Decide push (v int);

```
};
init (this *Process) Init () { this. status = PROCESS_IDLE; this. input = -1;
this. output = -1;
}
reaction (this $const * Process) initr (v int) { activate {
this.input = v;
if this.status == PROCESS_IDLE { this.status = PROCESS_ACCESS;
}
}
}
```

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

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

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

2)

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

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

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

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

moves the Process from the idle to the access state.

Thus, the state of the component logically flows from idle, to access, to decide, to done assumingthat the User does indeed activate the initr reaction.

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

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

The _init action shown in Figure 4.6 moves the User component from the request state to the waiting state while initializing the corresponding Process component via the initp push port. The _dummy action and the treatment of the error flag by the _init action are for consistency with .

The Decide reaction prints out the identity of the User and the value received by the User. If the User is not in an error condition and waiting for a response, then the value of the decision is recorded and the User changes to the done state. Otherwise, the error flag is set meaning that the corresponding Process activated the Decide reaction before the _init action.

Type UserStatus int; const USER_REQUEST = 0; const USER_WAIT = 1; const

```
USER_DONE = 2;
type User component { v int;
status UserStatus; decision   int; error bool;
initp push (v int);
};
init (this *User) Init (v int) { this.v = v;
this.status = USER_REQUEST; this.decision =   -1; this.error = false;
}
```

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

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

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

```
type System component {
```

```
    x Variable;
    process [3]Process;
    user [3]User;
};
init (this *System) Init () {
   this.x.Init ();
   for i... 3 {
        this.process[i].Init ();
        this.user[i].Init (i + 100);
   }
}
bind (this *System) _bind {
   for i... 3 {
        this.process[i].get_x <- this.x.Get;
        this.process[i].set_x -> this.x.Set... i;
        this.user[i].initp -> this.process[i].initr;
        this.process[i].Decide -> this.user[i].Decide;
   }
}
instance s System Init ();
```

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

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

port of each Process to the

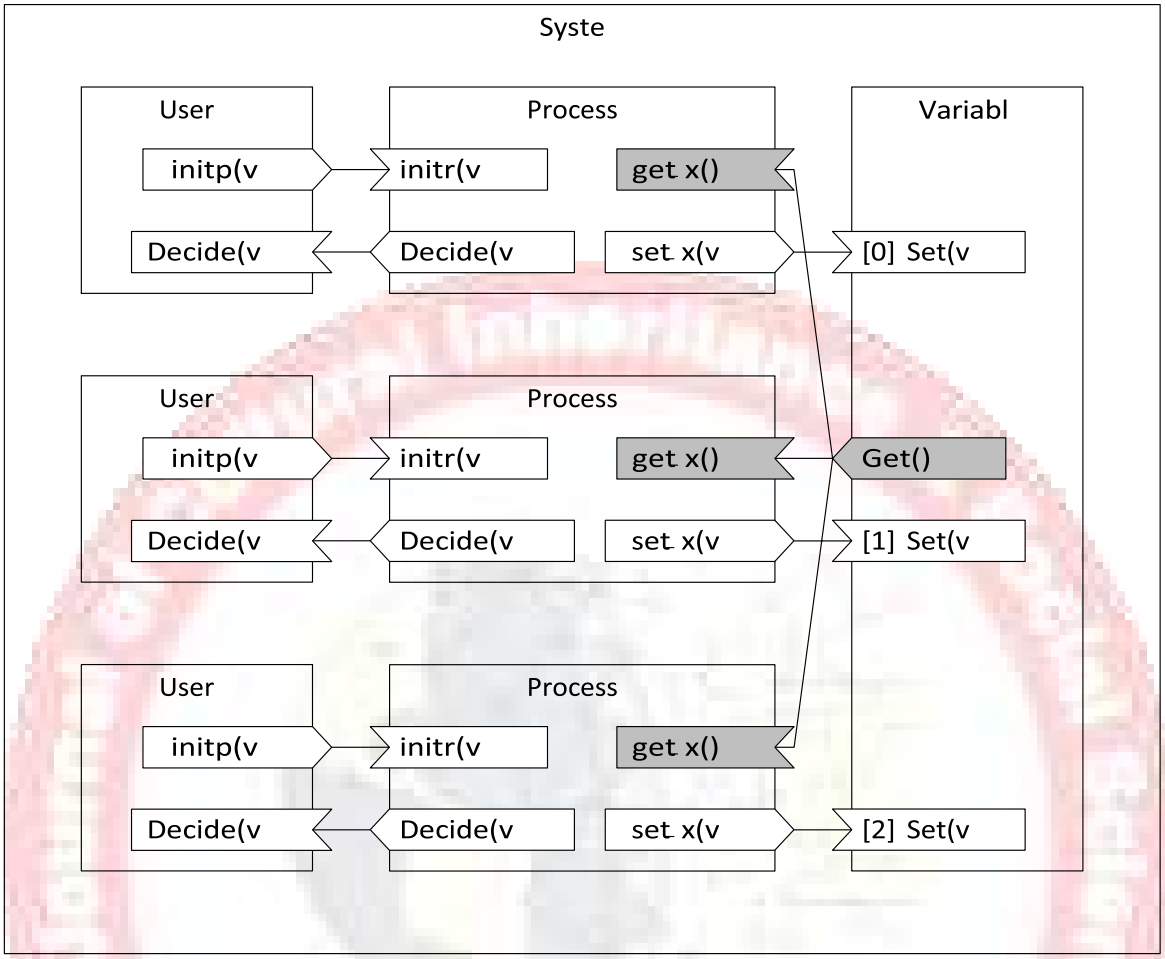

figure 4.8: Diagram of a System component of the Shared Variable System

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

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

x set to 102

0x1fd7258 decided value is 102

0x1fd71f8 decided value is 102

0x1fd7228 decided value is 102

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

```
type Channel component {
  queue Queue;
  receive push (message $foreign *heap uint);
};
reaction (this $const * Channel) send (message $foreign *heap uint) {
  var x *heap uint = move  (message);
  activate {
    this.queue.Push (x);
  };
}
action (this $const * Channel) _receive (!this.queue.Empty ()) {
  activate receive (this.queue.Front ()) {
    this.queue.Pop ();
  };
}
```

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

4. 6.2 Heap Channel System:

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

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

The send reaction moves the heap which makes the Channel the new owner of the heap. Any heap operations on message after the move statement will return nil. The

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

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

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

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

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

```
init (this *System) Initially () { }
action (this $const * System) _send  (this.send_count  !=  100)  {
  var x *heap uint = new (heap uint);
  change (x, y) {
    *y =  this.send_count;
  };
  activate send (x) {
    println (‘Sent ‘, this.send_count);
    this.send_count++;
  };
}
reaction (this $const * System) receive (message $foreign *heap uint) {
  var x *uint = merge (message);
  println (‘Received ‘, *x);
}
bind (this *System) Bind {
  this.send -> this.channel.send;
  this.channel.receive -> this.receive;
}
instance s System Initially ();
```

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

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

4. 7 Related Work:

rcgo is designed to be free from data races and draws upon existing work in this area. Common techniques include 1) augmenting type declarations, signatures, etc., to indicate sharing/locking requirements and effects; 2) associating objects with an owning thread or memory region; and 3) transferring objects from one owner to another. The foreign attribute of rcgo is similar to the lent attribute of Guava and the limited attribute of Promises . All three techniques allow a reference to be temporarily shared but not stored so as to create a shared resource. The transferable heap data type is similar to Islands and Values in Guava .

All three types represent object graphs with no external references which allow them to be transferred from one owner to another owner. In rcgo, objects are owned by component instances while Islands and Values are owned by threads.

Preventing data races in multi-threaded programs is accomplished by protecting critical sections with locks. Flanagan and Abadi developed a type system for statically checking for data races.

This formalism was used in the design of both Guava and Cyclone .

Locks are explicit in Cyclone while they are implicitly associated with the monitors of Guava. The reactive components of rc_{go} are similar to the monitors of Guava in that access to sharedstate is implicitly synchronized.

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

4. 8 Summary:

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

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

Conclusions

- A reactive system is characterized by "ongoing interactions with its environment" [66]. Asyn- chronous concurrency is a feature of reactive systems that makes them inherently difficult to develop.
- Reactive systems are already used in various forms of critical infrastructure and the number, diversity, and scale of reactive systems is expected to increase given the continuing proliferation of embedded, networked, and interactive systems. Decomposition and composition are two complementary techniques that are helpful when designing and imple- menting reactive systems, especially given such increases in complexity.
- We argue that the dominant techniques based on multiple sequential threads/processes thwart decomposition and composition and thus contribute to the accidental complexity associated with reactive system development.
- Specifically, we believe that a model for reactive systems should facilitate principled composi- tion and decomposition. Beyond defining units of composition and a means of composition, a model for reactive systems should facilitate practical techniques like recursive encapsulation and behavior abstraction through interfaces.

- Composition should be compositional meaning that the properties of a unit of composition can be stated in terms of the properties of its constituent units of composition. Finally, units of composition should be subject to substi- tutional equivalence meaning that the definition of a unit of composition can be substituted for its use and vice versa. Substitutional equivalence allows a complex system to be reduced to a single unit and a complex unit to be decomposed into a system of simpler units.

References

Clojure. http://clojure.org/.

Go. https://golang.org/.

Ioa language and toolset. http://groups.csail.mit.edu/tds/ioa/.

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

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

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

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.

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

G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent program- ming. ACM Computing Surveys (CSUR), 15(1):3–43, 1983.

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

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

David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of java

without data races. In Proceedings of the 15th ACM SIGPLAN Conference on Object- oriented Programming, Systems, Languages, and Applications, OOPSLA '00, pages 382–400, New York, NY, USA, 2000. ACM.

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

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.

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

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.

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

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

Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifi- cations for analysis and manipulation. In Proceedings of the 20th International Confer- ence on Software Engineering, ICSE '98, pages 167–176, Washington, DC, USA, 1998. IEEE Computer Society.

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

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

W.D. Clinger. Foundations of actor semantics. PhD thesis, Massachusetts Institute of Technology, 1981.

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

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

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

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

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

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.

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

M. Davis. Computability & Unsolvability. Dover Books on Computer Science Series. Dover, 1958.

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

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

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.

ECMA International. Standard ECMA-262 - ECMAScript Language Specification. 5.1 edition, June 2011.

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

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.

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

The Apache Software Foundation. http://www.apache.org.

The Apache Software Foundation. http://tomcat.apache.org.

D.P. Friedman and D.S. Wise. The Impact of Applicative Programming on

Multipro- cessing. Technical report (Indiana University, Bloomington. Computer Science Dept.). Indiana University, Computer Science Department, 1976.

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

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

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.

K.J. Goldman. Distributed algorithm simulation using input/output automata. Tech- nical report, DTIC Document, 1990.

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

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.

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

S. Halloway. Programming Clojure. Pragmatic Bookshelf, 2009.

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.

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