The Design of a Runtime System for Reactive Components

Justin R. Wilson

Abstract

The purpose of this document is to develop designs and evaluate alternatives for a runtime system for the reactive component model. In section 1, I review the reactive component model. In section 2, I develop the major constraints the model places on the runtime system. In section 3, I outline the basic architecture and different design dimensions. In section 4, I outline the design of a concrete implementation.

1 Reactive Component Model

Reactive components. A *reactive component* consists of state, actions, reactions, ports, sub-components, bindings, and accessors. The *state* of a component may be finite or logically infinite. An *action* consists of a *precondition* and *effect*. A precondition is a Boolean expression that indicates that the action can be executed. The effect specifies a deterministic update to the state variables. A *reaction* is a named effect. A *port* is a named interaction point used to link an action or reaction to a reaction. A reactive component may contain *sub-components*. A *binding* links a port to a reaction. An *accessor* synchronously reads the state of a component.

Execution. Computation proceeds by selecting a component-action pair, evaluating the precondition, and applying the effect if the precondition is true. Actions are selected non-deterministically and their effect is applied atomically. These tasks are performed by a *scheduler*.

Composition. Effects can conditionally *trigger* ports. If a port is bound to a reaction, then the reaction is applied with the triggering effect. Let (c, e) be a pair consisting of the component c and the effect e where the effect is either an action or reaction. We can define the relation $(c_0, e_0) triggers (c_1, e_1)$ if (c_0, e_0) may trigger (c_1, e_1). The transitive closure $Triggers(c, e)$ forms the *effect set* of (c, e). Computing the effect set for a component and action allows us to enumerate all of the components implied during the execution of an action. Thus, we refine the explanation of execution to say that all effects in the effect set of an action are applied atomically, i.e., the form a single atomic transaction.

The atomic execution of composed actions links the state and behavior of individual components and allows properties to be established between them. This facilitates compositional reasoning.

Two-phase execution model. For convenience, we divide the execution of an effect into two phases called the *immutable phase* and the *mutable phase* so named because the state of the component cannot change in the immutable phase while it can change in the mutable phase. In the immutable phase, a component may assume that all components in the system are immutable. A component is free to access the state of another component using accessors in this phase. Any references to a foreign component's state must be forgotten before entering the mutable phase.

2 Constraints

Constraint 1 (Non-interference)**.** The state of a reactive component can only be updated by its associated actions and reactions.

Constraint 2 (Deterministic execution)**.** The execution of each action must result in a unique next state for the entire system.

Explanation of constraints. The ultimate goal of both constraints is to facilitate the development of robust, predictable systems. The constraints guarantee certain behaviors which in turn allows programmers to reason about the systems they build. Non-interference allows a component developer to establish the properties (safety, progress) of a component from the text of the component as all state updates are evident from the text. The deterministic execution condition prevents a component from interfering with itself when composed.

Component analysis for deterministic execution. Logically, all of the immutable phases are executed before the mutable phases. An implementation is allowed to execute immutable and mutable phases in any way that enforces the semantics of the model and does not violate the constraints. For example, an implementation may execute the mutable phase of an effect as soon as all external references to its state are forgotten.

An effect is *potent* if its mutable phase might perform a state update. For deterministic execution, we require that there is at most one potent effect for a component in an effect set. We denote the corresponding analysis *component analysis*. Furthermore, we require that the *triggers* relation induce a tree structure as opposed to an arbitrary directed graph. A effect triggered by multiple ports (including cycles) may require that the state be updated in conflicting ways, hence, the structure is restricted to a tree.

Decomposition hypothesis. Component analysis does not consider that a component may be composed of discrete state variables. If we make this consideration, then deterministic execution requires that each state variable have a unique next value when executing an action. If the state of a component contains arbitrary linked data structures then verifying deterministic execution may require pointer analysis which is intractable in the general case. I hypothesize that if an action is deterministic under state variable analysis but not under component analysis, then the offending component(s) may be decomposed into additional components so that it is deterministic under component analysis. This decomposition may be automated.

Constraints and parallelism. The non-deterministic condition provides a way to loosen the assumption that the execution of each action be physically atomic. Consider all of the components implied by two actions. We can annotate each component as being read-only or read-write in the first action and read-only or read-write in the second action. The two effects sets are *independent* if the read-write components in each action are disjoint and the read-only components in the first action are disjoint from read-write components in the second action and vice versa. Independent actions can be executed in parallel.

Summary of constraints. The two constraints that must be met in a system for reactive components are non-interference and deterministic execution. Non-interference says that the state of a component can only be updated by the text of the component. Deterministic execution says that each action yields a unique next state. The constraints must always hold but can be established at any time between compilation and execution. These constraints are the driving force behind many of the design decisions.

3 Architecture and Design Dimensions

Basic architecture. A minimal runtime for reactive components consists of a *scheduler*. The scheduler, as indicated by the model, selects actions to execute. Time and space efficient policies for selecting an action and selecting independent actions for parallel execution are the major challenges. A *memory manager* is necessary if components have logical infinite state. The memory manager manages the allocation and deallocation of memory and the reservation of other resources like I/O ports, file descriptors, etc. A *loader* may be necessary if the system cannot be loaded directly by the operating system or if foreign components are loaded into the system. Similarly, an *interpreter* is necessary if the text of a component is not natively executable, e.g., virtual machine bytecode.

Hosted vs. unhosted. A *hosted* runtime is one in which the system of reactive components is executed as a process in a traditional operating system, e.g.,

Linux. The advantage of this approach is the hosting environment as it provides a substantial amount of infrastructure for the runtime. The drawback of a hosted runtime is the interface it imposes for interacting with an external environment. For example, processes in UNIX interact with external resources by making synchronous calls on file descriptors. A runtime for reactive components for such an enviroment would need to include file descriptors as an abstraction and functions for opening, configuring, reading, writing, and closing them. Most likely, the runtime would provide a built-in reactive component that wraps a file descriptor. The scheduler would include a Proactor to translate read-ready and write-ready file descriptor events into actions of the component wrapping the file descriptor.

An *unhosted* runtime is an environment where reactive components are the fundamental units of computation and form an operating system. The advantage and disadvantage of this clean-slate approach is that we are free to create our own conventions but must write every service from scratch. I assume that the unhosted runtime would target a 32-bit or 64-bit x86 processor. The runtime would include a built-in interrupt component that would translate physical interrupts into actions. The runtime and/or programming language would need to include support for assembly language for interacting with hardware.

I want to build an unhosted runtime because hardware drivers, protocol stacks, and the like will demostrate the full power or utter weakness of reactive components. However, that is a lot of work and I want to finish more than I want to see an unhosted runtime, therefore, I will pursue a hosted runtime. A good design should facilitate a transition from hosted to unhosted.

Open vs. closed. An *open* system allows foreign components to be loaded at run time whereas a *closed* system does not. In a closed system, determinism and non-interference *may* be verified at compile time. In an open system, they *must* be deferred to load time. I will pursue a closed system, however, the verification system will be a discrete entity that can be used in either the compiler or a loader.

Finite state vs. infinite state. A component may have finite state or have logical infinite state which requires dynamic memory allocation. For a first attempt, I will allow components with logical infinite state since this feature expands the kinds of systems we can build. Exploring optimizations for finite state systems should be pursued later.

Static vs. dynamic. A *static* system has a fixed number and configuration of components while an *open* system allows new components to be created and bindings to change. A static system *can* be verified at compile time while an open system *must* be verified at load time. I will pursue a static system.

Interpretation vs. compilation. The runtime may be designed around interpretted or compiled components. I will pursue byte-code interpretation

because 1) I think it will be easier to implement than a compiler, 2) it can be designed around the constraints of reactive components, and 3) it avoids the design of a high-level programming language. For example, we can include and exclude features that allow us to check that the state of each component is disjoint and remains disjoint.

Non-interference vs. communication. One way of enforcing non-interference is to place each component in its own virtual address space. The drawback of this approach is that communicating components must serialize and deserialize messages as they are sent which is tedious and error prone. Furthermore, it adds overhead every time a port is triggered. I envision systems of hundreds or thousands of communicating components in which case the overhead becomes unacceptable. The alternative is to place all components in the same address space which make communication easier and more efficient. Non-interference, then, must be verified through checking. The one-time cost of verification will be amortized over a possibly infinite number of sent messages.

Is it feasible to place all components in the same address space? I believe the answer to this question is yes based on the following observations. First, we have 64-bit processors which provide a huge virtual address space, one that is bigger that the physical memory available in any machine today. Second, the working set of modern systems fits in physical memory. (We do not operate computer systems so that they are swapping frequently.)

Serial vs. parallel schedulers. A *serial* scheduler executes one action after the other so they don't overlap in time. A *parallel* scheduler may execute actions so that they overlap in time. In addition to the analysis performed for deterministic execution and non-interference, a parallel scheduler requires analysis to determine which actions are independent. The analysis for independence is essentially the same as the analysis for non-determinism and can be done at compile time for a closed-static system. I will pursue a parallel scheduler.

Exceptions. A programming language may be restrictive enough to avoid exceptional conditions altogether. We may require that all exceptions that may be thrown must be caught. If this cannot be verified, then the runtime system must be designed to deal with exceptions. All exceptions can be related to the component in which they occur. One strategy, then, is to disable the component throwing the exception so that none of its actions or reactions are ever executed. This may result in a cascading failure, however, the presence of an unhandled exception indicates that the system was always unstable.

Pointers and memory management. I assume that pointers will be a first class object in the runtime. I still need to consider the implications of allowing pointers to members. If not, then the runtime only supports value semantics (like functional programming language) and garbage collection will be necessary

to manage the lifetime of values. Pointers must be managed to enforce non-interference. First, every pointer must be initialized to a sound address which is either null or the address of a valid object for that pointer. This can be accomplished by zeroing memory when it is allocated and type checking assignments. Second, every pointer must remain a sound address. This is accomplished by type checking assignments and preventing the formation of dangling pointers. Dangling pointers occur in systems with manual memory deallocation when the object to which a pointer refers is destroyed but the address of the object is retained. A dangling pointer would give a reactive component the ability to corrupt either the memory management system or another component if the object is reallocated. Consequently, the runtime system must use some form of reference counting or garbage collection. Third, pointers must be managed to keep the address space of each component disjoint and enforce non-interference. This can be addressed with type checking by only allowing reactions to receive copies or non-storable pointer types. Any address derived from a non-storable pointer cannot be stored.

Observations about garbage collection. There are two observations that may be useful in the design of the garbage collector. First, garbage collection may be performed on a component-by-component basis because the state of each component is independent of all others. If we assume mark-and-sweep, then marking begins with the component as the root and sweeping only considers the memory that has been allocated by that component. This suggests a two-level architecture featuring a global allocator and per-component allocator. Each per-component allocator acts like a cache to reduce the burden on the global allocator since the global allocator is a shared resource and must be synchronized for a parallel scheduler. Second, garbage collection can be thought of as an action. Thus, garbage collection can be performed during an action or as an independent activity. Garbage collection can be performed concurrently subject to independence. Performing garbage collection during an action requires that the stack be analyzed while performing it as a separate activity requires no such analysis.

Shared and transfered objects. An object is *self-contained* if the transitive closure of a points-to analysis forms a closed set identifiable by a single memory address. Self-contained immutable objects can be shared safely among components, although, such sharing is not required for correctness. However, if the immutable object is dynamically created, then there must be mechanisms in place to reclaim the resources associate with the object when it is no longer needed. This implies either a reference counting scheme or system-wide garbage collection. Reference counting is preferred since it keeps garbage collection from becoming a system-wide problem.

When triggering a port, a component may decide to transfer part of its state to another component. To maintain non-interference, we must ensure that 1) the transferred state is a self-contained object, 2) the transferring component

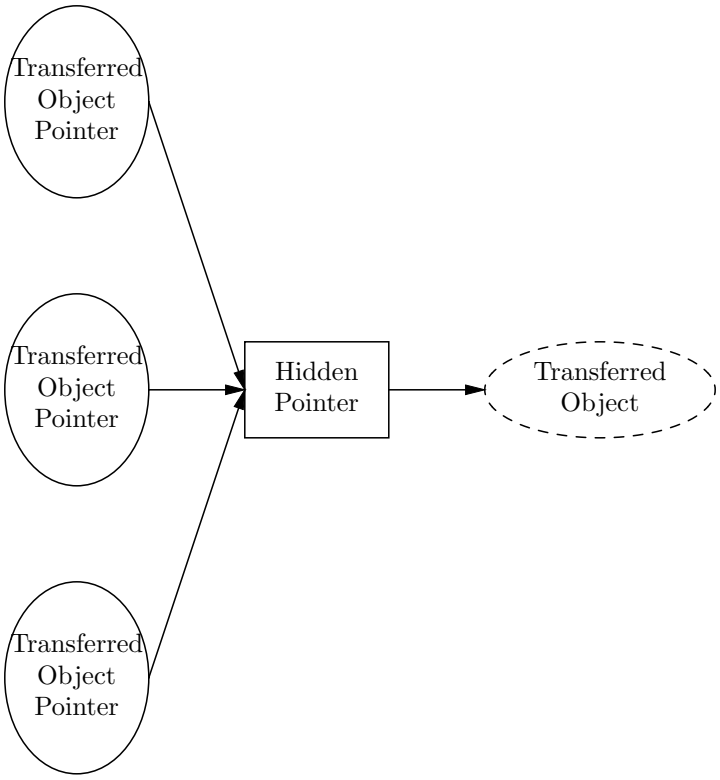

Figure 1: Pointer system for transferred objects.

retains no references to this object, and 3) at most one component receives the transferred object. If no component receives the object, the object should be reclaimed. Figure 1 shows how transfered objects may be implemented. On the sender side of the transfer, the hidden pointer is set to null while the hidden pointer on the receiver side is initialized.

Determining if an object is self-contained is difficult in the general case since it require pointer analysis. However, an object that neither accepts pointers in its methods nor produces pointers in its functions is self-contained.

4 Implementation

Virtual machine image. I propose an implementation of reactive components based on virtual machine images or system images. A system image contains sections containing the types used in the system, the state of each component, the code for each component, and list of bindings that defines the structure of the system. The user workflow consists of *loading*, *starting*, *stopping*, *saving*, *copying*, and *interogating* system images. A user may load a system image from a disk or other store. A user may then start the image which starts

the scheduler. A system may be executed for a fixed amount of time, until a fixed number of actions have been executed, or until interrupted by the user. The final system image may then be saved to disk, copied, archived, etc. The system image may be interrogated to reveal the current state of its constituent components. The overall goal of software development then is to prepare system images for use. System images may be edited directly or they may be produced through a conventional software development cycle, e.g., edit source files, compile, link, etc.

Loading. The loading phase is where the system is evaluated against the constraints of the model. The following conditions apply to the type section:

- no recursively defined types
- no undefined types

The following conditions apply to the state section:

- each chunk of memory has a valid type, matching sizes, etc.
- each pointer is sound (points to a object of the correct type or null)
- the state of each component is disjoint

The conditions for the code section will be discussed later. The following conditions apply to the bindings section:

- each component-action pair forms a tree
- a component has one mutable effect in an action tree

The listed checks are straightforward.

Code verification: immutable and mutable phases Effects must obey immutable-mutable phase semantics. A component cannot change state during the immutable phase. All port triggering must occur in the immutable phase. Both of these conditions can be verified using control flow analysis so longs as the implementation provides a clear separation between phases. Thus, one instruction may be devoted to signal the beginning of the mutable phase.

Code verification: isolation Isolation can be ensured by checking that no component retains a reference to the state of another component. We define four memory regions corresponding to the component's state, the stack for the currently executing effect, a heap consisting of objects allocated by the component during this effect, and the state of foreign components. Table 1 describes the access rights for pointers in the immutable phase. Table 2 describes the access rights for pointers in the mutable phase. The main points to observe are:

- the state of a component changes from immutable to mutable as expected

	Component	Stack	Heap	Foreign
Component	Immutable	No	No	No
Stack	Immutable	Mutable	Mutable	Immutable
Heap	Immutable	No	Mutable	No

Table 1: Allowed pointer relationships in the immutable phase. Each row represents the location of the pointer. Each column represents the access rights the pointer has into another region.

	Component	Stack	Heap	Foreign
Component	Mutable	No	Absorb	No
Stack	Mutable	Mutable	Mutable	No
Heap	Mutable	No	Mutable	No

Table 2: Allowed pointer relationships in the mutable phase. Each row represents the location of the pointer. Each column represents the access rights the pointer has into another region.

- permanent storage (component and heap) cannot store the address of an object allocated on the stack
- the component can absorb parts of the heap during the mutable phase
- pointers to the state of foreign components are only allowed on the stack and cannot be saved in permanent storage

So long as the runtime system can determine the phase and the origin of each pointer, it can check that the relationships in the tables are preserved. Accessors may allocate memory in the heap of the accessed component.

5 Code verification: shared objects

I am still evaluating shared objects. Currently, I feel that they add little to no value but will end up causing a substantial increase in complexity.

6 Code verification: transferred objects

Transferred objects do have value. For example, one can set up a pipeline that passes objects from stage to stage. Since the object is transferred, each stage can modify the object.