Software Architecture Using Fine-grained Event-driven Reactive Components

Paul Tarvydas
Visual Frameworks Inc.
tarvydas@visualframeworksinc.com

Norm Sanford
Visual Frameworks Inc.
nsanford@visualframeworksinc.com

ABSTRACT

In this paper, we describe design motivations and experience with a visual language that treats the architecture of a reactive system as a composition of small, asynchronous software components communicating via data-carrying *events*. This paper traces out the learning curve which ultimately concluded in the design of an *event-driven reactive* visual language and runtime system that facilitates construction of operating systems and applications completely in the event-driven paradigm.

The use of this system for developing distributed applications is discussed.

Categories and Subject Descriptors

D.1.7 [**Programming Techniques**]: Visual Programming; D.1.3 [**Programming Techniques**]: Concurrent Programming – *distributed programming*; D.4.1 [**Operating Systems**]: Process management – *concurrency*; D.4.7 [**Operating Systems**]: Organization and Design – *embedded systems*; D.2.11 [**Software Engineering**]: Software Architectures – *languages, patterns*; D.2.13 [**Software Engineering**]: Reusable Software – *reuse models*; D.2.2 [**Software Engineering**] Design Tools and Techniques – *state diagrams*; D.3.2 [**Programming Languages**]: Language Classifications – *concurrent, distributed and parallel languages, design languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features – *frameworks*

General Terms

Algorithms, Design, Reliability, Experimentation, Human Factors, Languages.

Keywords

Reactive, event-driven, components.

1. INTRODUCTION

Over nearly two decades, we have developed an event-based technology. We have implemented at least five variants of the technology – three of them as visual languages – and used them in industrial products. Most of the uses of the technology were in the field of *reactive systems* – e.g. embedded, distributed controllers and graphical user interfaces (GUI's).

DEBS '07, June 20-22, 2007 Toronto, Ontario, Canada

2. RELATED WORK

Our work resembles Flow-based Programming[11]. The technique described in this paper – Visual Frameworks™[16] – differs in that it allows multiple destinations for connections while preserving causality. Connections are strongly-typed at compile-time. The memory management schemes differ. We have emphasized visualization and compilation of diagrams to code. We have developed our technique to allow the paradigm to be used on small 8-bit CPU's and for building operating systems on "bare" hardware.

Communicating sequential processes[7] and OCCAM[12] employ named ports. Visual Frameworks™ uses near-first-class objects to organize connections between ports, removing this information from the components themselves.

Device monitors and deferred conditions[9] of Turing Plus constituted an early inspiration for the output pins and output queues of our work. Further inspiration was acquired from work on operating systems such as Tunis[8], Harel StateCharts[5] and Brad Cox'[3] concept of software IC's.

3. MOTIVATION

Our original motivation for this work came from the construction of a complex embedded controller utilizing four CPU's. The requirements posed a problem in that the scan-times for PLC's[1], the preferred solution, were too large. Context-switch times were too large to allow our problem to be solved using process-based RTOS's. Memory constraints and speed issues prevented the use of higher level languages and object-oriented programming techniques of the day, e.g. Eiffel[10] or C++[15]. The problem needed to be solved at the *driver* level (i.e. below the RTOS), yet the level of programming complexity was recognized to be fairly high (from a control-flow perspective). These issues caused us to search for structured approaches to writing the required *reactive* software.

A second motivation was our observation that hardware engineers using state-of-the-art technologies (e.g. TTL, VLSI) were able to create designs that were more predictable, reliable and testable than the designs produced by software designers using embedded software technologies of the day. For example, we observed:

1. that hardware was designed using *asynchronous* components

[1] Programmable logic controllers, using ladder logic.

2. that hardware components utilized *input pins* and *output pins* that provided full encapsulation and allowed *pin-compatible* interchangeable components to be *plugged* into circuits
3. that the *state machine* paradigm was well understood (e.g. formalized) and commonly used in hardware designs
4. that hardware designs were created using formal, well-defined diagrams (i.e. *schematics*)
5. that hardware component behavior could be easily tested and characterized using *test circuits*
6. that the hardware design paradigms ensured *causality*[2] (discussed in [16]).

With the above motivations, we proceeded to build a number of variants of *event-based reactive* languages. At least five of the variants were developed to production quality levels and were used to build commercial applications.

In most variants, we used the *event-based reactive* paradigm to program all aspects of the applications on "bare hardware" (i.e. no operating system) and we found simple ways to address all of the issues traditionally solved by RTOS's.

4. EVENT-BASED TECHNIQUES

4.1 Text-Based State Language

Our first attempt at building a multi-processor state-chart-like language was a textual language. The compiler detected all *active* variables – variables that might change due to external events (from simple drivers) or from internal events (state changes in other state machines) in expressions. The compiler emitted a global table of *active variables* vs. the optimum set of *active expressions* to be evaluated for every triggering of an *active variable.*

The method worked sufficiently well to solve the above embedded controller problem. The optimized table of *active expressions* ensured that external events were promptly serviced without incurring any scanning overhead. The overhead of fully preemptive context switches was replaced by less expensive interrupt-based cascades of state transitions.

We also implemented a GUI (graphical user interface) for the embedded system using the state language.

4.1.1 Insights

We gained a number of insights from this exercise.

We found that Harel's notion, at that time, of "micro-semantics"[6] was unsuitable for distributed, parallel execution – due to the requirement for synchronized micro-stepping of all state machines in the full system (including machines on other cpu's).

We found that Harel's concept of triggering state transitions based on transitions occurring in other, separate state machines led to programming and maintainability problems akin to those suffered with over-use of *global variables* in software – the cause of a transition was not explicitly visible on state-chart diagrams.

Implicit state-chart concurrency, in Harel's original notation, led to great complexity in the implementation of the language, e.g. the need to calculate transitive closures over the complete application to optimize the dispatch tables. We concluded that state machines and concurrency are completely different issues and should be treated separately in the language design.

The state-chart paradigm, used on its own to solve a complete 'real' problem, tended to prevent encapsulation.

The GUI-building problem was easily solved using the *reactive* paradigm, as opposed to other more common paradigms such as the *procedural* and *oop* paradigms.

Finally, we recognized a repetitive *pattern* in the code for delivering events between state machines and cpu's.

4.2 Events, Kernels, Input and Output Queues

In analogy to RTOS kernels, such as those described in [8] and sold as commercial products[14], we constructed an *event kernel.* We used the C programming language in conjunction with the kernel to program a commercial network controller / concentrator based on the Z80 8-bit micro-controller. The memory constraints (64K) precluded the use of more sophisticated off-the-shelf technologies.

4.2.1 Events, Kernels, Components, Drivers

We defined an *event* containing data to be three bytes – one tag byte and two bytes for data (i.e. large enough to hold a 16-bit pointer). A fixed-sized pool of *events* was statically allocated at compile/link time.

Program components each had their own *busy flag*, *input queue* and *output queue*. The project architecture was composed of thirteen such components. Memory for variables used by each component was statically allocated at compile/link time.

Device driver code was written as very short assembler routines. For example, the pseudo-code for character input drivers was:

1) read byte from port
2) reset port
3) call kernel to *send* the byte (allocate and event and enqueue to output queue)
4) call kernel to *exit* (restore interrupts, dispatch queued event).

Likewise, the kernel pseudo-code was:

1) field an interrupt, disable interrupts
2) look up device driver by index in a table and call it
3) on driver *exit,* restore interrupts and *dispatch* the event as described in steps 4-7
4) look up target component in a table
5) save interrupts, move event from output queue of sender to input queue of target, restore interrupts
6) call entry point of target component

[2] "A causal system is a system with output and internal states that depends only on the current and previous input values."[17], or – "WYSIWID" – what you see is what it does.

7) upon *return* from target component deallocate the input event then, if the target module's input or output queues contained events, dispatch them using steps 4-7 (and so on for further target components), else, execute a HALT instruction (become idle waiting for interrupts).

The system was completely interrupt-driven. CPU time was given to components by "threads" that were started when interrupts were fielded. The "threads" simply withered away when there was no more work to be done.

4.2.2 Insights from Simple Event-Driven System

This kernel design exhibited a desirable operating system property. It disabled interrupts only for very short periods of time – e.g. during steps 1-3 while fielding an interrupt and in step 5 while moving events between queues. It had a small memory footprint and rapid response to real-time interrupts.

Due to the inclusion of a *busy flag* and *input/output queues* the components acted like non-reentrant *processes*.

Non-reentrancy resulted in a number of benefits. The kernel was significantly smaller than RTOS kernels which supported pre-emption and context switches. Multiple process *stacks* were not needed – a single stack was used by the complete system. The application code turned out to be "simpler" than if it were written to use a full RTOS.

The concept of process *priorities* did not appear in the kernel. Priorities were completely controlled by hardware priority levels. We concluded that all software components should operate at the same priority level.

Non-reentrancy did not impede response to interrupts. It may have sped up response times, since full context switches were not required.

This design did not impede parallelism. Multiple threads were executed in a stack-like manner (last in first out) – following the hardware architecture of the CPU. Threads could be interrupted and the top-most thread on the stack would run to completion before a previous thread was resumed.

As long as the average rate of work could be handled by the CPU, the system naturally buffered and smoothed out bursts of high traffic, without adding complications to the device driver code.

The *busy flag* guaranteed that event order was preserved. During a burst of interrupt activity from the same hardware device, the first interrupt would begin a "thread" and execute component code. If another interrupt from the same source arrived before the first thread was finished, the second interrupt would simply queue up the event and quit – effectively being "blocked" by the (set) *busy flag* of the target component.

We found that device driver code – traditionally written in ad-hoc assembler or C – was better structured and modularized through the use of this system.

The strict and very simple definition of the inter-component communication mechanism (i.e. *events*) allowed several developers to work in parallel with high confidence that their code would integrate smoothly when completed.

At one point, we tried to "optimize" away the use of output queues and discovered that this led to unpredictable completion times for components, event reversal and breaking of causality. Output queues were essential to the implementation.

Finally, we found that this system validated the concept of using the *event* paradigm on bare hardware and that the restrictions – no preemption, non-reentrancy and cooperative multitasking – posed no irritants nor roadblocks to the design of the application. We demonstrated that commercial applications could be constructed using this combination of techniques.

4.3 Pins, Parts, Hierarchical Schematics, Wires, Visualization

The next variant of the *event* technology introduced a number of concepts, many of them borrowed from the digital hardware CAD paradigm. The terminology we use was also borrowed from the well-established terminology of the hardware world.

4.3.1 Parts

We defined the concept of a small, asynchronous software module with *input pins* and *output pins* and called these modules *parts. Parts* perform *reactive actions* of some kind and have an *input event* API and an *output event* API. In contrast to the *components* of the previous variant, *parts* could have multiple input and output *pins*.

There are two kinds of *parts.* Hierarchical parts are a composed of other parts and the *action* performed by the hierarchical parts is dependent on the arrangement of the composed parts. Code parts are leaf nodes that contain no other parts and contain executable code (in some text or visual language).

All *parts* also own a *busy flag*, an *input queue* and an *output queue*.

Parts have unique names. *Parts* can be *instantiated* any number of times within a system in a manner reminiscent of *classes* and *objects*, although *parts* provide no notion of *inheritance* and instead use *composition*

4.3.2 Typed Pins

We defined two kinds of pins – *input pins* and *output pins. Parts* are allowed to have any number of *pins*. *Parts* with no pins are considered to be *top-level parts* (i.e. the complete application(s)).

Furthermore, we assigned a *pin number* (an index) to each pin, a pin name to each *pin* and a *type* to each *pin.* The *pin type* denotes the *type* of the data to be transmitted on the pin.

4.3.3 Wires

We defined the concept of a *wire* to be a conduit in which *events* flow.

Wires connect *output pins* of *parts* to *input pins* of *parts.* There is no restriction on the number of connections or where *wires* must go – except that all *pins* connected to the same *wire* must be of the same *type* and every wire must be completely local to a single *schematic.* For example, a *wire* might connect the *output pin* of a *part* to the *input pin* of the very same *part.* Additionally, *input pins* and *output pins* may have no connections ("N.C.") to *wires* at all – in which case the N.C. *input pins* are never triggered and *events* sent to N.C. *output pins* are ignored.

4.3.4 Hierarchical Schematics

Systems are *composed* of *parts*. The mechanism for such composition is called a *schematic*.

Schematics contain *part* instances and *wires*. *Schematics* may contain *parts* that are *schematics* themselves – this allows a system to be architected in a hierarchical manner.

Since *schematics* are a type of *part,* they also own a *busy flag,* an *input queue* and an *output queue.*

4.3.5 Visualization

The above decomposition of software modules into *parts, schematics* and *wires* lends itself to semantically complete diagrams of systems.

We originally chose to represent *parts* by any arbitrary 2D graphic and / or by icons.

Parts have two representations. The *outline* represents the outward appearance of the *part* including its name, its *pins* and its iconic symbol. The *implementation* represents the "insides" of the part – other *schematics* or code.

Pins were originally represented by small squares with the *pin names* displayed to the side.

Wires are represented as lines. Originally, each wire *type* was encoded as a different color of the line.

A sample visual *schematic* (from a modern version of the tools) appears in Figure 1.

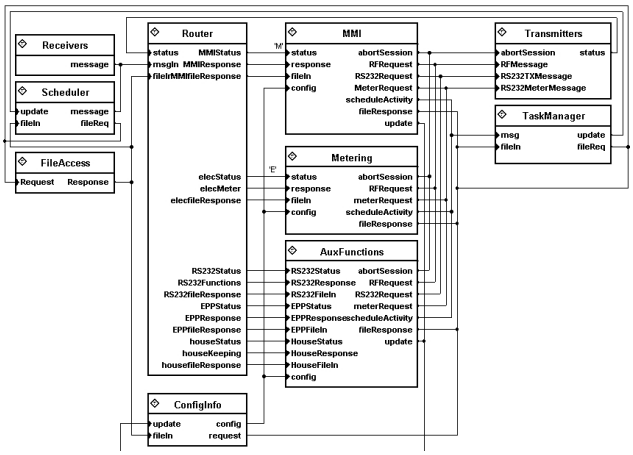

Figure 1. A sample schematic.

This technology was used to construct an IDE (integrated development environment) running on Windows® for a commercial development system in the point of sale industry. The final IDE system used about 500 unique *parts* and was composed of 20,000 *part instances* and used the C language as the internal language of *leaf parts*. About 4 developers built the entire system.

4.3.6 Insights from the First Visualized Event System

This was the first system in which diagrams of the system were an integral part of the design process and toolset.

In retrospect, it appears that the addition of visual editing to the third generation of the *event-driven* paradigm brought an explosion of ideas similar to that of Stephen J. Gould's concept of "punctuated equilibrium"[4]. The basics of *event-driven* architecture were laid in the previous phases. Adding diagrams and diagram compilers allowed us to try out a large number of ideas. Later, in subsequent phases, we culled the ideas and produced more "tame" tools and technologies that were used in a number of commercial applications.

Firstly, we validated the concept that diagrams could be stongly-typed and compiled just like textual source code. The concept was proven to be applicable to 'production quality' industrial systems. The system was an extreme form of cooperative multitasking, employing 20,000 small "processes", i.e. 20,000 instances of event-coupled parts, many of which contained only one line of code or an expression. Despite this fact the system performed efficiently and was indistinguishable from a Windows® program written in ad-hoc C code.

The first impulse of designers working in the *event* paradigm using diagrams as source code, was to attempt to construct "everything" as parts. Some designers went so far as to create expression operator parts – e.g. a '+' part which added two inputs and output a result. The resulting diagrams were too detailed and reduced understandability of the designs. This tendency towards overly-small-grained parts was also driven by a compulsion to reuse as much "code" as possible. From this we concluded that there is a distinct line between "architecture" and "code" and that an ideal toolset would incorporate both – diagrams for architecture and textual languages for code. We concluded that reuse of "code" was much less important than reuse of "architecture" - clear designs and diagrams resulted in more effective units of reuse.

We found that diagram "style" was important and that the field of electrical engineering was a good resource for how to draft software diagrams. Diagrams that followed lay-out rules similar to those of "schematics" in electronic CAD systems (e.g. wires should be drawn only in horizontal or vertical directions), turned out to be the most readable, understandable and maintainable.

We also experimented with memory allocation. In this variant, all non-scalar data was allocated internally within parts. If the non-scalar data was sent as an event, the part would provide a "return" pin so that the non-scalar data could be returned to the originating part for recycling / discarding. The concept worked, but created a number of inconvenient issues. The diagrams, which represented the architecture of the system, were cluttered with architecturally uninteresting memory management minutia. The system's principle of strong typing led to a plethora of "return" pins on parts – one for each returnable non-scalar type. The memory management "return" pins became semantically aliased – used for more than one purpose. Along with being simply drop points for discarded data, the return *events* were used for synchronization *events* (acknowledgments). Synchronization was architecturally interesting, but was obscured when aliased with return events.

A repetitive pattern emerged – a large majority of parts were implemented as state machines (event reactors). We found that most parts contained small snippets of code and the "glue" code to turn these snippets into viable event reactor parts often

overwhelmed the “useful” code in the parts. We concluded that this repetitive pattern should be automatically generated by the compiler system and should not be hand-written by the programmers. No off-the-shelf language lent itself to compiling small snippets of code. These observations pointed to the idea that the system should provide an inner language for which we had better language-level control (e.g. no pointers) and runtime control (e.g. a virtual machine based language).

Lastly, this variant demonstrated that the *event-driven reactive* paradigm was well-suited to building GUI applications.

4.4 Downloadable Parts, VM, State Diagrams

In the fourth phase of the technology, we constructed a CAD-like IDE for constructing software using the *event* paradigm.

We included a new class of diagram – the hierarchical state machine. The hierarchical state machines did not include concurrent states, since concurrency was already addressed by schematics and parts. The state machine entry, exit and transition code 'snippets' were written in a text language (an application specific language based on a virtual machine supporting garbage collection).

The target application was a series of commercial embedded controllers based on the 8-bit 8051 cpu sold into the point of sale market. The small RAM space (13K-32K writable RAM, 128K-512K write-protected code space) precluded the use of traditional RTOS methods and processes with dynamically allocated stacks. The use of static state machines allowed us to predict and constrain RAM usage.

One goal of the design was to allow in-field, remote upgrades to the software. We accomplished this by storing wiring tables and instance data in RAM and the code for parts in code memory.

The event kernel was extended to perform 'wiring' at boot-up time. The binary images for parts contained a small, recognizable sequence of 'header' bytes at the front of the image.

Part images were downloaded via RS232 by the kernel and inserted into code space.

On boot-up, the kernel performed the following actions:

1. The kernel searched all code space for the unique headers and made an inventory of all parts residing in memory in a manner similar to early versions of OS/9[14].
2. The kernel recursively instantiated all *top-level* parts – i.e. parts that had zero pins. Note that more than one top-level part could simultaneously co-exist in a system.
3. To instantiate a leaf (code) part, the kernel allocated instance data for the part as required and an output pin array for the part.
4. To instantiate a schematic part, the kernel recursively instantiated all parts required by the schematic, then allocated a wiring table and filled it with connection lists. It filled in the output pin arrays for the instantiated parts with pointers (indices) to nets.

Once the production units were shipped, they could be updated in the field by a technician or a customer. The 'downloader' application was, itself, written as a schematic. The 'downloader' application was invoked by resetting the terminal and entering a certain keystroke sequence during boot-up. The downloader communicated via RS232 with a server to determine the minimum set of parts to be upgraded, via download, overwriting previous versions of the same parts. After downloading new parts, the terminal rebooted and the kernel performed the boot-time wiring sequence as above, picking up and wiring the newly downloaded parts in with the rest of the system.

This system greatly decreased download time. Instead of taking some 15 minutes to download an entire new copy of the software, the downloadable / pluggable part technology reduced download time to minutes (sometimes seconds).

The average part size was around 1,000 bytes for parts containing VM instructions. The smallest part we created was 6 bytes long – 4 bytes for the header and 2 bytes for an integer constant. The part was a “constant emitter” part. The application was parameterized by constants that could be downloaded.

4.4.1 Insights

This system was used for many years (approximately seven) to develop a number of point of sale applications.

The combination of *schematic* diagrams for expressing architecture and *hierarchical state machine* diagrams for expressing inner implementation was shown to be productive.

We observed that developers with training in electronics naturally understood the *event-driven reactive* paradigm – creating systems using hundreds of small “processes”.

A repeatable pattern emerged for the design process, itself. Applications were architected by a group of designers, at the *schematic* level on a white-board and transcribed into the toolset. The visual tools checked the type compatibilities of all pins and wires, providing a kind of “daily build” feedback during the architecture phase. Errors would be immediately repaired and the architecture tended to stay “integrated” during its development. In this application domain, architectures tended towards having about two-hundred to three-hundred parts. Following this architecture phase, the full set of parts was listed on a Gantt chart and the developers estimated development times for each part. Estimates for any part exceeding three days caused the architects to decompose the part into further smaller parts[3]. Following this, a larger group of developers implemented and tested the inner code for all parts in parallel. Final “integration” times were relatively small, since much of the integration had been performed during architecture.

For this class of application, the number of architects tended to be about three, and the implementation group tended to use five to ten people (always including the original architects).

The notation provided for *driver parts* – parts that were implemented in assembler and directly handled the hardware I/O ports. Driver parts generated the interrupt threads that ran the system.

---

[3] In practice, architects developed an intuition for when a part was “small enough” and this decomposition cycle was no longer necessary.

Two patterns of driver part usage emerged. Some architects preferred to place driver parts at the top level of the architectures, whereas others preferred to hide the driver parts at the bottom of architecturally interesting hierarchical black boxes. We discovered that, to support driver parts contained within non-driver parts, the kernel needed to recognize events originating within a driver and to set the *busy flags* for all containing / parent parts in the tree up to the top level.

The diagram editors were "smart" editors. The editors would not allow, even temporarily, inconsistent connections nor inconsistent types. Repeated use of the tool set showed that this behavior made certain drawing and editing operations very difficult for users. This indicated that languages using diagrams for code should mimic the well-established techniques used in textual language compilers. Editors should be "free form", leaving error checking for subsequent scanner, parser and semantic analysis compilation passes. We also found that editing diagrams as source code requires a richer set of gestures than is provided in common 2-D editing programs (Paint®, Visio®, etc).

We found it desirable to allow for system initialization using events. The basic kernel algorithm was modified to contain three states (1) creation and initialization, (2) top-down dispatching of events issued during initialization, and (3) the "steady" state, enabling interrupts and responding to external stimuli.

4.5 Diagram Compilers, Testing, Background Processing

The fifth variant involved a complete rewrite of the runtime system and VM for cross-platform portability and a significant rewrite of the diagram editor and compilers to decouple editing from compilation. This system was used to develop a smart electricity metering technology.

4.5.1 Insights

A number of practical testing techniques were explored and used. One of the variants of the portable kernel was a workstation (Windows®) version. Coupled with scripts and GUI's written in Tcl/Tk[13] the application was substantially developed and tested on a network of workstations in parallel with the development of the target hardware. The interfaces to the driver parts for the application were specified during the architectural phase. Two sets of "plug compatible" drivers were developed – one set that communicated with the networked simulators and one set for the actual hardware. When the hardware became available and most useful simulation-based testing was complete, the developers simply unplugged the simulation drivers and plugged in the real drivers and continued testing on the hardware. The majority of the system had already been tested and required no further re-testing.

The application required some long-running low priority processing. The kernel was extended to give CPU time to a system of "background" parts when there was no interrupt activity, delivering events to background parts in a round-robin fashion.

A simple and effective path test method was employed. The test personnel created simple test jigs (using schematics and parts) for every part in the final application. Then they simply marked, using a highlighting pen on printed copies of the state diagrams, all of the executed transitions as they tested the parts. They modified the test jigs to force all transitions to be taken. Management found the marked-up hard copy to be an effective measure of testing progress.

The ability of the system to reuse parts and to refactor architectures was validated. After about five person-years' of work had been spent creating and testing parts, the business case changed dramatically. A system satisfying the new requirements was produced in about two weeks by refactoring the top-level architecture and modifying a small number of parts. All remaining parts were reused.

Finally, we found that it was practical to treat diagrams as factbases[2] containing simple graphical relationships (lines, points, text) and to use Prolog-like[1] inferencing to "parse" the facts into parts and wiring relationships, then to proceed to compile this information using traditional compiler techniques.

5. EXPERIENCE

5.1 New Paradigm, New Patterns

We found that an *event-driven* architecture allowed the use of a number of 'new' *software design patterns* that had been borrowed from the digital hardware paradigm.

The *daisy chain pattern* was developed to explicitly and visually resolve certain priority issues. In traditional *process* and *thread* paradigms, such issues are resolved invisibly by the kernel. Parts were placed on the *schematics* in left-to-right order and wires carrying a token event connected left-to-right, as shown in Figure 2. The left-most part had the highest priority. If the *part* decided to handle the event, it would do so and not pass the token on. If the *part* decided to ignore the event, it would *send* the event to the next highest priority *part* in the chain, to the right.

The *arbiter pattern* was developed in conjunction with the *daisy*

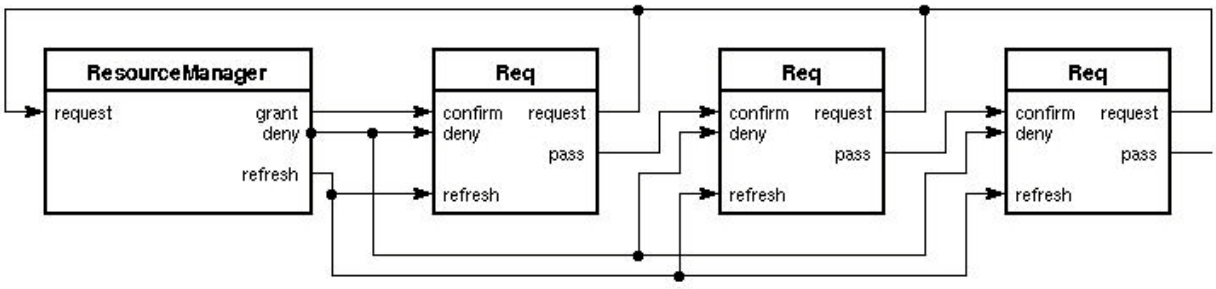

Figure 2. Daisy chain architectural pattern.

chain pattern to assist in resolving priority issues and resource allocation issues. The *arbiter part* was copy-pasted into all of the parts in the daisy chain. A version of the *arbiter pattern* is shown in Figure 3 and Figure 4.

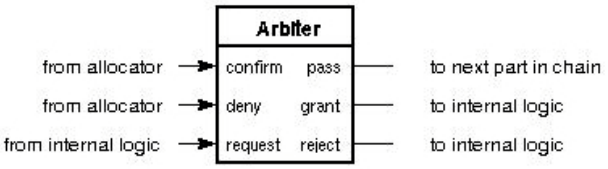

Figure 3. Arbiter part used inside daisy chained parts.

Each *part* in the daisy chain contains an *arbiter part.* When the daisy chained *part* decides that it needs a certain resource, it *sends* one event, on the same wire, to two places – the request pin of its enclosing schematic and to the request pin of its own *arbiter.* The *event* contains an identifier that specifies the desired resource. The *arbiter* arms itself by moving to its *waiting* state. The request pins of all parts requesting resources are tied to a *bus* wire that it is connected to a resource controller part. The resource controller metes out access to the resources by sending reply events down a *daisy chain*.

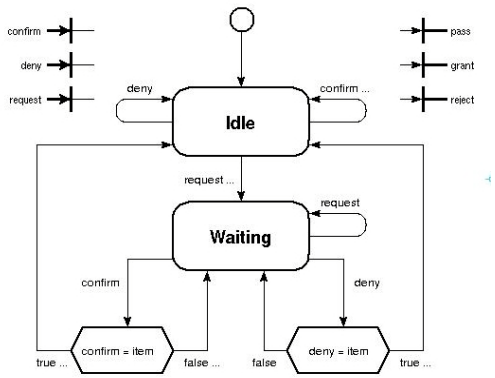

Figure 4. Arbiter implementation state diagram.

The replies can be in two forms and are received on two different pins:

1. Resources that are granted arrive on the *confirm pin.*
2. Resources that cannot be granted arrive on the *deny pin.*

Since all parts are reactive and asynchronous, the timing of the *grant* or *deny* events is completely controlled by the resource allocator part. It may wish to delay its response knowing that the requested resource will become available at some future time. It might wish to deny a resource when it knows that the resource will never be renewed or when it determines that the resource will not be renewed in a sufficiently 'short' period of time.

The arbiter, within requesting parts, responds to events on its *confirm* and *deny* pins. Since different parts may have requested different resources, it is possible that grant / deny events for resources that are 'uninteresting' to the arbiter may be seen. The arbiter simply forwards 'uninteresting' events further down the chain by sending them out of its appropriate forwarding pins. When the *arbiter* part receives a *confirm* or *deny* event involving the resource that it has been armed for, it seizes the event and does not forward it downstream. It sends the event to sibling parts on the same enclosing schematic via its *grant* or *reject* pins.

These daisy chain and arbiter patterns are similar to *super* calls in OOP hierarchies. The use of *events* makes it possible to clearly design and document these relationships on diagrams.

5.2 Programming Distributed Systems

The *event-driven reactive* paradigm begins with the assumption that all parts are distributed, i.e. that every part runs on its own CPU. For "N" parts and "M" cpu's, N = M. When it is not possible to use M cpu's, we *simulate* multiple cpu's. When only one CPU is available, M = 1.

Systems designed in the *event-driven reactive* paradigm have the property that they can be designed to be distributed across multiple processors, or, they can distributed after the fact. Event sending implies no synchronization between parts. The impact of distributing event-driven parts across multiple processors is significantly lower than trying to do so with synchronized call-return based software.

We described the concept of *downloadable parts*. This is a simple form of distributed processing and was a natural outgrowth of our creating systems using the *event-driven reactive* paradigm.

We described the distributed test jigs that were used on projects. The application requirements did not make it necessary to run tests across LANs. The developers simply executed the tests over a LAN because it was more convenient to do so, without recognizing that they were doing something considered to be more complicated in other paradigms.

In a discussion of distributed programming, the issue of exceptions must be addressed. Distributed systems employ protocols to detect normal and exceptional behavior. In the *event-driven reactive* paradigm, exceptions are no different from other data, i.e. exceptions are asynchronous events. This observation was borne out by the above work – most of the above commercial applications required the implementation of protocol based communications to / from other distributed computers. We have found that it is natural to architect such distributed systems by creating protocol parts (depth of hierarchy depending on the complexity of the protocols). We then simply included these protocol parts on appropriate schematics. The remainder of the application simply sends events to the protocol parts without knowledge of which protocol is being used or whether the communication is local or networked, similar to the test strategy above.

As discussed above, we found it easy to *re-factor* architectures that were implemented in the *event-driven reactive* paradigm. Using visual tools, the act of refactoring architectures often devolves into *cut-paste* operations of regions of the diagrams, creation of some new schematics, new parts and re-wiring.

6. FUTURE DIRECTION

6.1 Notation for Distributed Computing

We hope to explore whether slight notational changes would make distributed applications appear more transparent on the diagrams.

The notation could allow assigning a "CPU" property to parts, denoting which cpu they are to run on.

If *wires* were considered to be simply another kind of hierarchical part containing an 'implementation' (i.e. a protocol), the wires could be made to send events locally or to other cpu's depending on the kind of wire used in the architecture.

Another example for consideration is visual grouping of parts and assigning a cpu / computer to the grouping.

6.2 Dynamic Load Balancing

We hope to explore visual design of dynamic load-balancing across multiple cpu's / multicores.

A number of possible patterns for performing load-balancing can be imagined.

The *multiplexer* pattern, Figure 5A, could distribute work-load events to similar applications running on different cpu's.

The *shunt* pattern, Figure 5B, could be used to break an application up across multiple cpu's and change the assignment 'on the fly'. Note that if a part receives no *events*, it remains idle. The *shunt* parts could simply redirect events to other cpu's, effectively side-stepping the shunted parts. The shunted parts do not move between cpu's – they become idle due to lack of input events.

The *kernel* and *wires* could be altered to allow dynamic routing of events to different cpu's.

The *enable / disable* pattern, Figure 5C, could be used to dynamically disable certain parts on different CPU's. When disabled, the parts go into a state that ignores all incoming events.

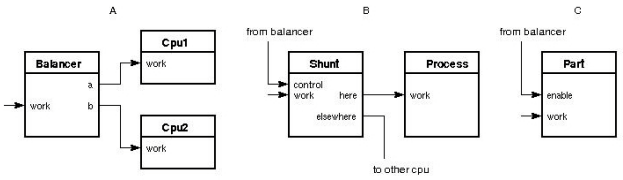

Figure 5. (A) Multiplexer. (B) Shunt. (C) Enable.

7. CONCLUSION

We discussed the evolution of an industrial quality *event-driven reactive* paradigm and visual languages and tools that support it. We discussed experience gained from building industrial applications in this manner.

We conclude that one very effective way to build event-driven systems is to construct such systems entirely in the *event-driven reactive* paradigm – "from the ground up". We have shown that it is possible to do so and that the overhead in terms of resource usage, speed and development effort is competitive with other state-of-the-art software development methods.

8. ACKNOWLEDGMENTS

This research was supported by the National Research Council Canada Industrial Research Assistance Program (NRC-IRAP).

9. REFERENCES

[1] Clocksin, W. F., Mellish, C. S. *Programming in Prolog*. Springer-Verlag, 1984

[2] Cordy, J. R. *Evolution of the LS/2000 Software Architecture*. tarpit.rmc.ca/cficse/2000/lectures/gl-cordy.pdf

[3] Cox, B. *Object-oriented Programming; An Evolutionary Approach*. Addison-Wesley Publishing Company, 1986

[4] Gould, S.J., *Wonderful Life: The Burgess Shale and the Nature of History*. Replica Books, 1989

[5] Harel, D., Statecharts: a visual formalism for complex systems. *Science of Computer Programming, vol. 8, pp. 231-274,* 1987

[6] Harel, D., Pnueli, A., Schmidt, J.P., Sherman, R., On the format semantics of statecharts (extended abstract). *Proc. 2nd IEEE Symposium on Logic in Computer Science*, IEEE Press, New York, 1987

[7] Hoare, C.A.R. Communicating sequential processes. *Communications of the ACM, vol. 21, pp. 666-677*, Aug., 1978

[8] Holt, R. C. *Concurrent Euclid, The Unix System, And Tunis,* Addison-Wesley Publishing Company, 1983

[9] Holt, R. C. 1988. Device management in TURING PLUS. *SIGOPS Oper. Syst. Rev. 22, 1 (Jan. 1988), 33-41*. ACM Press, New York, NY, USA

[10] Meyer, B. *Eiffel The Language*, Prentice Hall International (UK) Ltd., 1992

[11] Morrison, P., *Flow-Based Programming: A New Approach to Application Development*, Van Nostrand Reinhold, NY, 1994.

[12] OCCAM® 2.1 REFERENCE MANUAL. SGS-THOMSON Microelectronics Limited, May, 1995

[13] Ousterhout, J. K. *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, USA, 1994

[14] http://www.radisys.com/products/Microware-OS-9.cfm

[15] Stroustrup, B. *The C++ Programming Language*, Addison-Wesley, 1985

[16] Tarvydas, P., Sanford, N., Software Architecture with Visual Frameworks. *Canadian Conference on Electrical and Computer Engineering*, IEEE, 2006

[17] http://en.widipedia.org/wiki/Causality