SOFTWARE ARCHITECTURE WITH VISUAL FRAMEWORKS™

Paul Tarvydas
Visual Frameworks Inc.
email: tarvydas@visualframeworksinc.com

Norm Sanford
Visual Frameworks Inc.
email: nsanford@visualframeworksinc.com

Abstract

We present an overview of a software architecting method and visual language we call Visual Frameworks™. The method creates concrete, strongly typed software architectures through composition of concurrent software components.

Keywords*: visual language; software components; reactive; software architecture; asynchronous*

1. Introduction

Visual Frameworks™ (aka VF) is a language for expressing software architectures together with the implementation of software components.

We have built practical compilers for Visual Frameworks™, which compile the architectural syntax and semantics to executable code.

The syntax of Visual Frameworks™ consists of hierarchical diagrams. The architectural diagrams resemble electronics schematics, where *parts* (software components) are represented as boxes with *pins* and where inter-part data paths are represented as *wires* (directional lines).

Wires carry *events* between *parts*. Events are simple data structures, which travel in one direction only - no call-return protocol is used for inter-part communication.

Parts have both, input and output interfaces. They receive events on their input pins and send events from their output pins. Parts have no knowledge of where input events come from nor where output events will go. The wires on the enclosing schematic specify all event routing.

VF has a number of interesting properties:

- parts are pluggable and can be replaced and upgraded in the field by "pin compatible" parts
- parts can be instantiated multiple times in the same project
- parts are parallel, asynchronous and highly encapsulated
- interconnections between parts are described (only) by *schematics* which are, themselves, fully pluggable parts
- fine-grained concurrency employing only a single stack
- VF preserves causality (time ordering of events), a property which leads to intuitive
- the system has a small footprint and small computational overhead, making VF suitable for systems as small as 8-bit controllers
- parts can be specified and tested using "test jigs"
- reliability and manageability of projects is enhanced
- software "integration" can be performed early in the project, before the "coding" phase
- VF facilitates communication between project managers, architects, programmers and customers.

2. Overview of Visual Frameworks™

VF is a hybrid of diagrams and textual code. All VF diagrams are semantically complete. All code, including diagrams, is compiled to produce executable programs.

The philosophy of VF is to use graphical notation where it can beneficially express concepts that cannot easily be described with text, such as architectural details, and to use text where it is the most appropriate notation.

VF does not try to capture all programming concepts in its visual notation. We have stripped the architectural aspects of programming down to a very few simple constructs – *parts, events,* and *schematics.* A simple way to view the VF concepts is to consider VF as a system of *communicating state machines.*

VF is not a modeling language. Instead, VF expresses architectural concepts and design details in a way that is directly compilable to executable code.

VF was inspired loosely by Harel StateCharts[1]. We found that Harel's notation and micro-semantics[2] prevented true parallel "run-to-completion" operation of reactive systems. In practice, we found that it was more convenient to provide a notation for concurrency that is distinct from the notation used for hierarchical state machines. VF hierarchical state machines are a structured and non-concurrent version of StateCharts.

In VF, concurrency is expressed through the concept of *schematics.* A sample *schematic* appears in Figure 1. Furthermore, s*chematics* express encapsulation, composition, architectural hierarchy and inter-part communication.

A VF system is hierarchically composed of software modules, called *parts*, interconnected by one-way data paths called *wires*.

Schematics are, themselves, a type of *part*.

In our current version of VF, there are two other types of *parts – hierarchical state machine* parts (example in Figure 2) and *native code* parts.

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

3. Structure

3.1. Parts

Parts represent the fundamental unit of encapsulation in VF.

Parts contain executable code in either diagrammatic or textual form.

The contained code cannot communicate beyond the borders of its enclosing part other than through the use of *input* and *output* pins that form the input and output *event* interfaces of the enclosing part.

Parts cannot determine where their input events originate, nor where their output events are sent – e.g. the code within a part only "sees" events arriving at input pins and "loses track" of events once then have been sent to an output pin.

Parts are implemented as (a) hierarchical schematics, (b) hierarchical state machines or (c) textual code.

Parts process input events in the order in which they arrive. A *part* responds to an input event by performing a computation that may change its state and/or generate one or more output events. Event processing is atomic – a *part* processes an input event to completion before processing the next event. Output events generated by the processing are not propagated until the process is complete.

Parts are fundamentally asynchronous – they run independently of one another and "at their own speed". More than one part may be active in the system at the same time. In cases where synchronization between parts is desired, the software architect must explicitly design in such functionality.

3.2. Schematics, Wires and Causality

A composition of *part* and *wire* instances is called a *schematic.*

Schematics resemble network diagrams, where delivery of events is guaranteed and, thus, require no explicit network protocol.

Part and wire instances are encapsulated by a schematic and are not visible outside of the schematic – namely to the schematic's parent, its siblings and its children.

Wires are event-paths that connect *output pins* of parts to *input pins* of parts. *Wires* may contain multiple branches – more than one *output pin* from more than one part can "source" *events* onto a wire. Likewise, more than one *input pin* of more than one part can receive ("sink") events from the same wire.

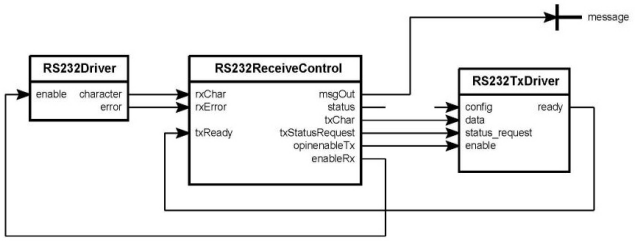

Fig. 1. Sample schematic.

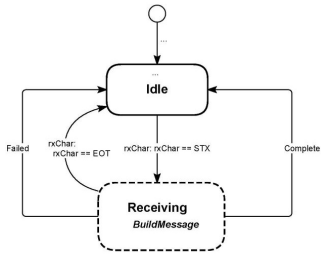

Fig. 2. Sample hierarchical state machine.

VF guarantees causality – time ordering of events. Events on the same wire arrive at all input pins connected to that wire at the "same time" and in the same order. For example, consider a wire connected to the output pin of part W and to an input pin of parts X, Y, and Z. If W generates an event *a* followed by an event *b,* then all three parts X, Y and Z will see an *a* before they see a *b.*

VF explicitly defines that there is *no* time-ordering relationship between *different* wires. In the above example, part Z might have another input pin connected to a source which sends *c* events. Part Z will still see the *a* event before it sees the *b* event, although it might receive zero or more *c* events *between* the *a* and *b* events.

3.3. Events

Events are simple units of scalar or structured data, tagged by an input- or output-descriptor.

When a unit of data is *sent* to an output pin, an *event* is created. The event contains the output pin descriptor (e.g. a part-dependent output index, or an output name) and the data itself.

When an event arrives at an input pin, it contains an input pin descriptor (e.g. a part-dependent input pin index or a name) and the data.

Output events are mapped into input events by the containing schematic (a part itself), based on the wiring information contained – only – within that schematic.

3.4. Informal Semantics

The semantics of VF can be best understood by defining parts to contain input pins, output pins, a single input queue, a single output queue and a "busy" flag.

At the beginning of time, a part is "idle" - it has no input events in its queue, its busy flag is cleared and it is executing no code.

When an event arrives at any input pin, the part is activated – its busy flag is set, it removes the first event from the input queue and begins processing some action that is associated with the input pin identifier. Events that arrive while a part is "busy" are simply queued (in order of arrival) on the input queue. During processing of a single event, the part may choose to *send* events to any of its output pins. These events are queued on the output queue of the part. When all processing for a single input event has been completed, the part performs two actions: (a) it checks the input queue for further

events – if the queue is non-empty, the part repeats the above cycle; if the queue is empty, the part resets its busy flag and becomes idle, and, (b) it releases the contents of its output queue to its parent schematic. The order of these two actions is not defined by VF – it is simply a implementation / scheduling decision.

A schematic operates in a similar fashion, except that its "processing" step consists of mapping output events to input events between the parts that it contains. A schematic is "busy" as long as any part contained by the schematic is "busy". In order to preserve the causality principle, a schematic must perform mapping and distribution of a single output event – to all of its sinks – in an atomic manner.

3.5. Hierarchical State Machines

State machines are parts that (minimally) contain *states, container states, terminal states* and *transitions. State machines* have *initially* and *finally code. States* and *container states* have *entry code* and *exit code. Transitions* are triggered by input events filtered by optional *guard expressions* and execute optional *transition code. Container states* each contain a single (hierarchical) sub-machine. *Terminal states* have *entry code* and can only be found in sub-machines.

Sub-machines are akin to subroutines – a machine "calls" a sub-machine by transitioning to a *container state.* The sub-machine "returns" to the containing machine by transitioning to a *terminal state.*

Transitions from a container state can be triggered by (a) events or (b) termination of the contained sub-machine, in which case the names of the sub-machine's terminal states can be used to select distinct transitions.

Transitions are inherited and cannot be overridden by sub-machines. If a container makes an event-based transition, then all of its sub-machines are forced to exit. This form of operation is vital to the principles of VF – every diagram tells a complete, encapsulated story. Its sub-machines – which appear in separate, unrelated diagrams – cannot override the operation of a state machine.

3.6. Kernel

A small *kernel* effects the transfer and mapping of events from the output pins of parts, to the enclosing schematic and to the sibling receiver parts. In other words, the kernel interprets schematics and state machines.

In practice, the kernel also includes a virtual machine to interpret the textual code parts. The use of a virtual machine ensures better encapsulation. We have, also, experimented with textual parts written in C, assembler and Common Lisp – in which case it is the designers' responsibility to ensure that rules of encapsulation are not violated.

The kernel usually includes a loader. On reset, the loader searches for parts in memory and then creates tables / code to instantiate and link the part instances as described by the schematics. The loader is simple and small, because it only needs to locate parts, instantiate them and create interconnections between pins (which, in the optimized case, are simply indices into a vector). The simple loader co-resides with the kernel. The loader is based on schematics – it simply needs to wire output pins to input pins. These simplifications allow us to dynamically update parts in the field – i.e. download a new part, reset and reload – with much less effort than required by traditional linkers and loaders.

3.7. Compiler

The compiler operates in a fairly standard fashion, except that its scanning and parsing phases need to recognize graphical entities, since the source language is largely graphical. At present, we use Prolog-like[3] inferencing to identify relationships between graphics entities (lines, arcs, text, etc.) in the scanning and parsing phases. The parsing phase produces a textual representation of the system – after which, well established compiling techniques are used.

4. Impact of VF Semantics

4.1. Causality – Cause and Effect

"A causal system is a system with output and internal states that depends only on the current and previous input values."[4]

Consider the seemingly trivial system in Figure 3.

An external event, *a*, is applied to the system at pin ip0 and is transmitted to pin ip1 of part X and pin ip4 of part Y.

The event *a* causes part X to generate an output event *b* on output pin op2 which is transmitted to pin ip3 of part Y.

If the diagram is implemented using call-return, it is possible that the events will arrive "backwards" at part Y.

In a call-return based scenario, a straightforward implementation would call part X first, with *a* as a parameter. X would then call Y, with *b* as a parameter. Only after X returns, would Y be called, with *a* as a parameter. In this "straightforward" implementation, we get the "unexpected" result that Y sees a *b* before it sees an *a* – i.e. the diagram belies what really happens.

In fact, the order of events at part Y is implementation-dependent using call-return. If we were to call Y first (with *a*) then call X (with *a*) would we see the "expected" behaviour (Y sees *a*, then *b*).

As a consequence, call-return semantics make this diagram ambiguous and worse, misleading. Such unexpected behaviour

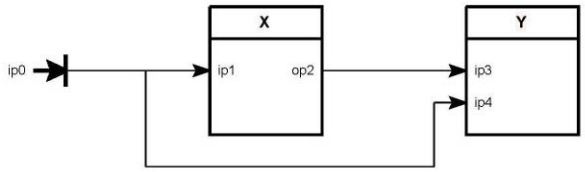

Fig. 3. Schematic demonstrating need for causality.

can be considerably more difficult to identify in non-trivial cases.

For this reason, previous attempts at drawing straightforward diagrams of software systems have met with little success.

VF semantics guarantee that this system will behave in an intuitive manner, that is, the events will always arrive at part Y in the correct order.

Consequently, VF can be used to draw intuitive diagrams of software architectures.

4.2. Output Pins and Event Mapping

Another key feature of VF is the fact that the event paths are separated from and opaque to part implementations. Parts have no knowledge of what lies beyond their borders.

Non-schematic parts cannot explicitly invoke other parts anywhere in the system.

Schematics can instantiate parts by name and shepherd events between only those parts they own. Schematics cannot see any other parts in the system, beyond those that they instantiate. This is in contrast with procedural and object-oriented languages, which typically require the source code of the caller to explicitly name the callee or a method.

VF enforces a communication hierarchy between parts, by virtue of the fact that schematics control all communication. Events can only travel "up" or "down" the hierarchy by entering or exiting the pins of schematics.

Schematics are first-class objects. Like any other part, schematics are pluggable, hence, the wiring between a set of parts can be changed by simply plugging in a new schematic.

Pluggability is facilitated by the fact that VF parts have, both, input pins and output pins. The concept of output pins appears in only a few paradigms in software, for example in the idea of networks, in process-based message sending and in a few languages such as CSP[5] and Occam[6].

One of the consequences of pluggability and opacity is that a system can be changed (repaired, upgraded) by plugging in a single new part. This change can be effected at long distance.

Pluggability, opacity and hierarchical composition also make it possible to draw intuitive, structured architectural diagrams. In fact, what we call "architecture" consists of the visual placement of parts and the visual routing of events.

These features allow VF architects to replace parts with equivalent parts easily and with little impact on the system, even when the parts have different internal implementations.

5. Conclusion

VF facilitates drawing of intuitive architectural diagrams that are independent of implementation details.

VF diagrams are semantically complete. Diagrams compile to code. There is no need to model code.

Encapsulation and output pins lead to the concept of testability via "test jigs". Test circuits are simply schematics whose purpose is to provide stimuli to the input pins of the part under test and to monitor the responses from the output pins.

VF facilitates remote upgrading of programs because only the modified parts need to be downloaded. In view of the fact that parts are typically small (hundreds of bytes) this is a significant advantage in systems with low-bandwidth connections or a large number of deployed units.

VF architectures are built using many – hundreds through thousands of – small, well-defined components. It is easier to estimate the implementation time for small, well defined, encapsulated components. In addition, since the standard error is inversely proportional to the square root of the number of parts, large numbers of parts lead to more accurate estimates.

Communications between architects, programmers and project managers are facilitated by the fact that VF diagrams are intuitive.

The VF paradigm also affects software "integration". The bulk of VF design work consists of creating a set of hierarchical schematics that describe the system to be implemented – i.e. the "architecture" phase of a project. The system of hierarchical schematics can be compiled and checked for consistency before any further code is written. This process is the same as the "integration" phase of most software projects, except that it occurs early in the project – the "architecture" phase – instead of late in the project.

Testability is enhanced due to the fact that wires and state machine transitions are visualized. For example, a VF user found a low-tech way to perform coverage testing for most of a VF system. The user simply printed out the diagrams and used a highlighting pen to mark paths that had been visited. The marked-up diagrams served as a good tool to communicate to the project managers how testing was proceeding.

We believe that application of VF to multi-processor environments will be a fruitful area for future investigation.

References

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

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

[3] W.F. Clocksin, C.S. Mellish, *PROGRAMMING IN PROLOG.* Springer-Verlag, 1984

[4] http://en.wikipedia.org/wiki/Causality

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

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