Software Wrappers for Rapid Prototyping JAUS-Based Systems

Bill Smuda, US Army TARDEC

ABSTRACT

Recent experiences with robots in Iraq have proven that robotic technology is useful to the warfighter, but tools are needed to rapidly respond to evolving missions. This paper details a methodology for automatic generation of software wrappers using JAUS to simplify prototyping and development of robotic systems (distributed, embedded and real-time system software modules). Software wrappers will allow insertion of modules into a visual prototyping environment. The wrappers will intercept module functions and bind them with functions needed to exercise the modules outside of the native environment. Automatic generation of JAUS wrappers will enhance the development environment by reducing rote work and producing consistently behaving module interfaces. The resulting methodology will provide a rapid prototyping environment for use in sensor integration, Operator Control Unit (OCU) development and autonomous vehicle control.

Keywords: Software Engineering, Robotics, Rapid Prototyping, Software Generation, JAUS

1. Introduction

The Department of Defense and the Army have invested heavily in automation infrastructure to aid in the development of defense systems. High Performance Computing assets have been distributed across the country and connected by a high performance computer network, the Defense Research and Engineering Network (DREN). The Army's Simulation and Modeling for Acquisition, Requirements and Training (SMART) program is expected to reduce cost, development life cycle and improve the war fighter's edge.

Similarly, a large investment is being made in robotics, both for current operations and especially for Future Combat System (FCS). Both current robots and FCS rely on distributed assets communicating over wired and wireless networks, albeit on different scales.

Payoff from these initiatives can only be realized if we can develop the reliable, high quality software that will allow material developers to conduct large tradeoff analysis in a reasonable time frame.

The current state of software development industry is not amenable to the lofty goals of the above programs. Software continues to be largely developed by hand. Simulation and prototype analysis require the direct involvement of expensive and scarce software experts. Engineers that develop software for individual components are often experts on the technical domain and thrust into the software development arena by necessity. In many cases domain experts have little knowledge of system and network administration. Forcing them to learn these additional disciplines dilutes their knowledge base and takes time away from their primary responsibility.

1.1. Automation. To improve the software development picture we need to discover place where we can automate the process. Tremendous gains have been made by stepwise automation of the manufacturing process. Automotive paint robots have moved human operators out of the paint booth and to a control panel. The result is consistent, high quality finishes on automobiles. The paint station operator no longer paints the car. A specialist configures the paint station for the current assembly line. The operator is presented with a limited number of operational choices on a control panel.

Similarly, in the software development we need to find places where we can automate parts of the process that are difficult and done by hand. We need to discover high payoff domains that can be automated in a re-configurable fashion. One of these domains is building the network configuration and instrumentation for prototyping distributed systems.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **01 MAR 2005**	2. REPORT TYPE **N/A**	3. DATES COVERED **-**
4. TITLE AND SUBTITLE **Software Wrappers for Rapid Prototyping JAUS-Based Systems**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **Bill Smuda**		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **USA TACOM 6501 E 11 Mile Road Warren, MI 48397-5000**		8. PERFORMING ORGANIZATION REPORT NUMBER **14730**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S) **TACOM TARDEC**
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
U.S. Government Work; not copyrighted in the U.S. Presented at SPIE Defense & Security Symposium 2005, Orlando, Fl USA, The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**SAR**	**9**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

The inherent complexity of computer network environments has limited the potential gains of using distributed prototyping and simulation in the material development process. The network configuration parameters (ports and addresses) needed to allow distributed elements to communicate are often difficult to create and reuse. Firewalls and separate firewall administrators not included in the design environment confound the situation. An incomplete understanding of network standards and operation scenarios confound the domain experts attempting to prototype a distributed system. Ad hoc methods reduce repeatability and make comparing alternatives difficult if not impossible.

1.2. **Objective.** The objective of this work is to describe a methodology that includes automatic generation of software wrappers to simplify development of distributed, embedded and real-time robotic systems. This work can also be extended into the more general case of communicating heterogeneous distributed systems. Automatically generated wrappers will allow insertion of modules into a prototyping environment. The wrappers will intercept module functions and bind them with functions needed to exercise the modules outside of the native environment. Unlike the well-known security wrappers, prototyping wrappers will be tailored to the current state of the prototyping environment. Automatic generation of the wrappers will enhance the development environment by reducing rote work and producing consistently behaving module interfaces.

2. Wrappers

2.1. **Wrappers.** Wrappers are executable programs that facilitate interoperability among existing standalone programs and/or program modules that were not explicitly designed to communicate with each other. The wrapper consists of at least two parts, specific information about the module it is wrapping and an interface to the external world. In the case of wrapping distributed elements, the interface to the world consists of specific information about the common network communication environment. The wrappers allows the distributed prototype to treat each of the elements as a component. In reality, in terms of data flow, we are creating programs to for each of the arcs in a data flow diagram.

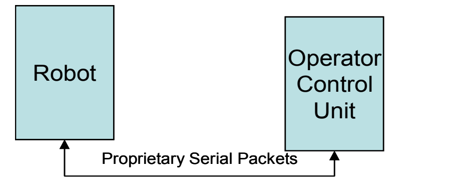


Figure 1 A Simple System

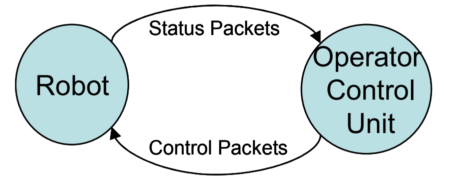


Figure 2 Simple System Data Model

Consider the following simple example. We have a robot and a Operator Control Unit (Figure1). The two components communicate via a serial interface. The interface carries proprietary packets. In the simple case, we are interested in determining how the system will work under different conditions, using Ethernet instead of serial for instance.

We note that the system can be modeled as a dataflow diagram, a graph

$$G = (V,E)$$

Where V is the set of Vertices and E is the set of Edges. Each vertex is an operational element of the system, and each edge is a data stream (Figure 2).

Next, we remodel the system as in (Figure 3). In this case, we break the direct serial connection between the robot and the Operator Control Unit. We connect the robot serial line to a device at the robot end and the Operator Control Unit (OCU) serial line to a device at the OCU end. The new devices intercept the original packets and translate them to messages.

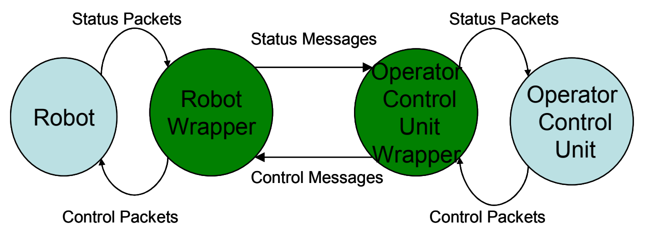
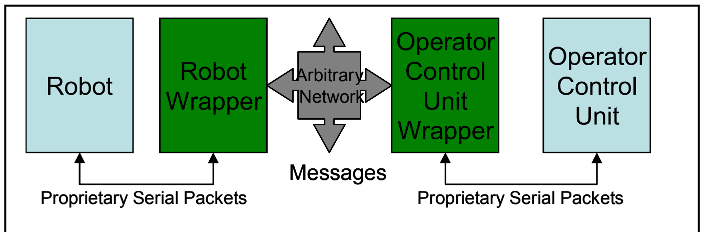


Figure 3 Simple System Data Model with Wrappers

When the system is implimented, it can now communicate over an arbitrary network, or communication device, that is dependent on the code in the wrappers. Note however, that the communications code for the two wrappers must match over the dataflows between two communicating wrappers.



Robot
Robot Wrapper
Arbitrary Network
Operator Control Unit Wrapper
Operator Control Unit
Messages
Proprietary Serial Packets
Proprietary Serial Packets

Figure 4 Simple System with Wrappers - Implementation

We can now model the system as an Extended Hypergraph

$$G = (V,E, W(v))$$

Where $W(v) = (T(v), \{A(v,e)\}, C(v,e))$

V is the set of verticies, E is the set of edges, T(v) is a translation, {A(v,e)} is a set of aspects related to the execution of the prototype and C(v,e) is the communication mechnisim to be used on on a particular edge.

Now consider a situation where we have a working robot. What happens if it runs into something? Obviously, from the above models, we can easily see that unless obstacle detection is a built-in function of the robot, nothing will happen. We next consider the case where we want to extend out simple robot prototype with a bumper.

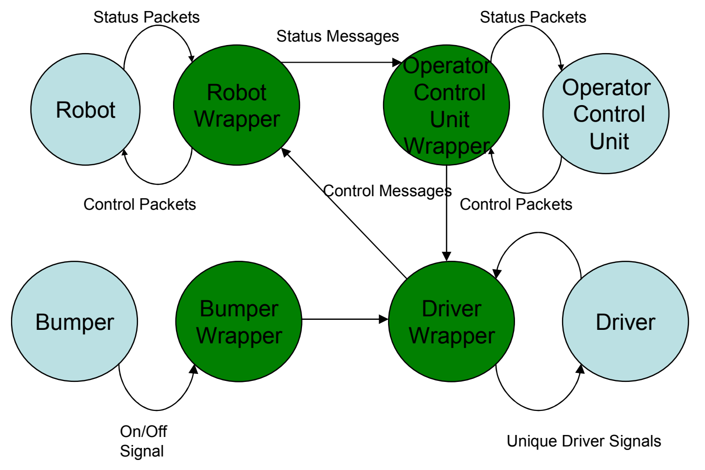


Figure 5 Modified System Model

In order to extend the prototype with a bump stop device, we may discover that the neither the robot or the OCU can accept a bump message intelligently. We might have found this out by initially having the bumper wrapper generate a stop message and send it directly to the robot the probable result would a robot that will not move until it is physically removed from the obstacle. The more robust solution requires that we find or create an additional component, a driver, to accept the messages from the OCU and from the bumper. Logic in the driver component then makes the decision to send an OCU command forward or to send one or more messages to the robot to stop or take corrective action.

As in the the prototype system descripion language (PSDL[1]), V represents an operator. In this case, the operator will almost always be a hardware in the loop module or a simulation of a hardware module. Each node V can only be entered via a wrapper, so for simplicity, since each node has the wrapped module hidden from the person assembling the prototype, we simply model the graph with the endpoint module and its wrapper as a single node in the graph (Figure 6).

$$G = (V', E)$$

Where V' represents a node V with function W(v) applied.

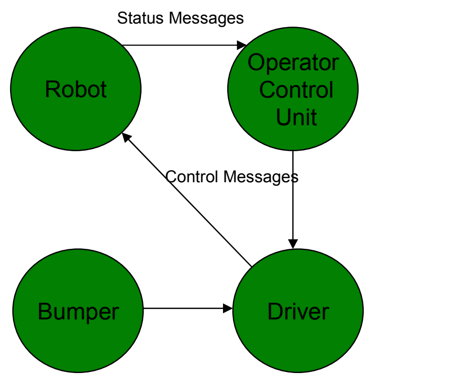


Figure 6 Simplified Notation

The ability to add functionality without modifying existing elements is known as the " Paradigm of Independent Extensibility"[2]. The principle function of component orientations is to support this principal.

2.2 Components. In many cases, the engineers who are ultimately creating the prototype are not software engineers. Mechanical, electrical and other engineers have learned to rapidly prototype systems using off the shelf components for the most part and custom assemblies of off the shelf components to create a hierarchy of components resulting in a breadboard or brassboard prototype.

Mechanical engineers have access to dozens of catalogues offering gears, struts, pumps and actuators, just to name a few. Electronics engineers have TTL manuals, discrete chips, a range of resistors and capacitors as well as more complex integrated circuits. In both cases, the specifications for the components are usually very clear and concise. The interfaces are not always simple, but they are immutable. An and gate has 2 inputs and one output, a 24 tooth gear has 24 teeth; there is no changing the interface, and obviously no reason to do so.

On the other hand, software engineers do not have it as easy. In a 1999 an article in Software Engineering, [2] boldly announced "The emergence of software components as a viable approach to software development represents a maturing of the discipline". Here it is 2005 and I doubt if I could find one mechanical or electronics engineer who could point me at a catalog of software components. I imagine I could find one if I looked hard enough, but my guess is that it would be business software components, not engineering.

The reasons for this are many, but probably mainly economic. Mechanical and electrical engineers prototype with "things". If a mechanical engineer needs 7 gears in a component, 7 gears are purchased. If a electronics engineer over voltages a chip and burns it out, a new chip is purchased. The reality of software is that it is not purchased in discreet parcels. Either one has to purchase a license to a complete collection (possible cost prohibitive) or the vendor has to sell inexpensive bits and pieces with no guarantee of return business (not a good business model).

The issue is confounded even more when one attempts to find a definition for a component. The distinction between a component, a class and an object is not always made clear. In his book "Component Software" [3] Clemens Szyperski describes a component as having the following three characteristics:

- Is a unit of Independent deployment
- Is a unit of third party composition
- Has no externally observable state

This definition, would be expected if we acquired and used software components the way we might prototype with TTL chips.

Fortunately, there are other definitions of components that relax the requirement of "third party composition". Czarnecki and Eisenecker refer to components as simple building blocks combinable is as many ways as possible[4].

From this definition, we can refer to the endpoints, the individual hardware items or simulations as components for the purposes of this paper. Therefore, wrappers are software, that at a minimum, provides a mechanism to allow components to be connected. Extending the wrappers, they can also contain an arbitrary number of aspects (within experimental bounds) to instrument the prototype, induce disturbances, simulate communications protocols, throttle communications speed and provide translations.

2.3 Joint Architecture for Unmanned Ground Systems (JAUS). From the JAUS website[5]:

"JAUS is mandated for use by all of the programs in the Joint Robotics Program (JRP). This initiative is to develop an architecture for the Domain of unmanned systems. JAUS is an upper level design for the interfaces within the domain of Unmanned Ground Vehicles. It is a component based, message-passing architecture that specifies data formats and methods of communication among computing nodes. It defines messages and component behaviors that are independent of technology, computer hardware, operator use, and vehicle platforms and isolated from mission."

JAUS is chosen for this work for the above reason, as well as the fact that it is in the process of transitioning to an Society of Automotive Engineers (SAE) Aerospace Standard.

The JAUS approach is to mitigate the “Tower of Babel” effect that is inherent when a number of independently created components are brought together. The typical approach for building a prototype robot system is for all the intermediate code to be created by hand. In most cases, time is of the essence, so shortcuts are taken. In some cases, the code is written by students, and is driven by an assignment or thesis requirement. In most cases, after the prototype is completed and examined, the knowledge behind the code evaporates. If a new question or project arises, most times, the process begins from scratch. The goal is to capture the knowledge gained during the exercise and make it reusable. Creating point to point translations requires a maximum of n(n-1)/2 translations[6] to make each node communicate with each other. Using an intermediate representation, such as JAUS requires a maximum of 2n translations to convert from the source to the intermediate representation and then to the destination representation.

JAUS is of course not the only intermediate representation. There are other messaging schemes in use, the important part is to have a consistent messaging architecture.

3. Prototyping

There are many reasons create a prototype:

- One might want to investigate and gain a better understanding of the requirements.
- One may need to understand a physical attribute that is too complex for calculation.
- There may be a compelling need that must be addressed quickly, while normal development progresses in parallel.
- One may subscribe to a spiral development model, where each successive prototype adds additional functionality.

In any case, a common attribute of a prototype is that it is not the complete and final solution. Typically, prototypes we create are used in a limited test. There is something we need to understand. Therefore, prototypes should be amenable to instrumentation. We need to be able to measure some aspect of the system, either internal or external. In either case, we would like to present a simple interface to the engineer or technician. We would like to be able to capture information, both about the configuration of the prototype and the results of running it.

3.1. Phases of a Prototype.

3.1.1 Preperation. The preparation phase formalizes the process of creating a prototype. Rather than create the entire prototype by hand, engineers create the interfaces to the components of interest and additional aspects that may be needed to execute a prototype, most commonly instrumentation. These small packages, possibly components in their own are stored for future use. The minimum set includes the code to translate the interface of the component of interest into a standard representation, and an interface to the world, usually a network interface. Along with the code, a set of instructions might be created specifying compilers, installation instructions and instructions on how to start the component if necessary.

3.1.2 Setup. In the setup or assembly phase, the prototype is built from the elements created or installed in the preparation phase. This is accomplished using a graphical user interface (GUI) to connect the components The components cannot communicate directly, so each component is associated with a wrapper, which can be customized during the assembly phase. After a graphical model is created, parameterized and error-checked, a set of wrapper programs is created. The wrapper programs should have attached documentation needed for their deployment. The assembled prototype should be stored for future reference and configuration management.

3.1.3 Run Time. When the prototype is operated, various aspects of the system will be captured and stored for future analysis. There is also the possibility at this point to monitor the run-time instrumentation in real time. By doing so, we can display various conditions to the test team as the prototype execution continues. Real time constraints described in the instrumentation can be monitored and an indication of how closely the system is operating in regards to timing bounds. Network traffic can be monitored and the capacity of a particular link displayed as a color change on the graph's edge.

3.1.4 Analysis. Data collected during run time is retrieved with the assembled prototype for analysis.

3.2. Prototyping Environment.

Examining a feature model based on the above four phases of a prototyping (Figure 7) one finds several areas ripe for automation (outlined in dashed lines). However, most are optional features in the sense that they are not terribly difficult to do by hand, or they already require the involvement of an expert. The mandatory feature labeled compiler, is the feature that creates the wrappers. This is where we would like to embed the knowledge of the software specialists for use by expert in the hardware domain.

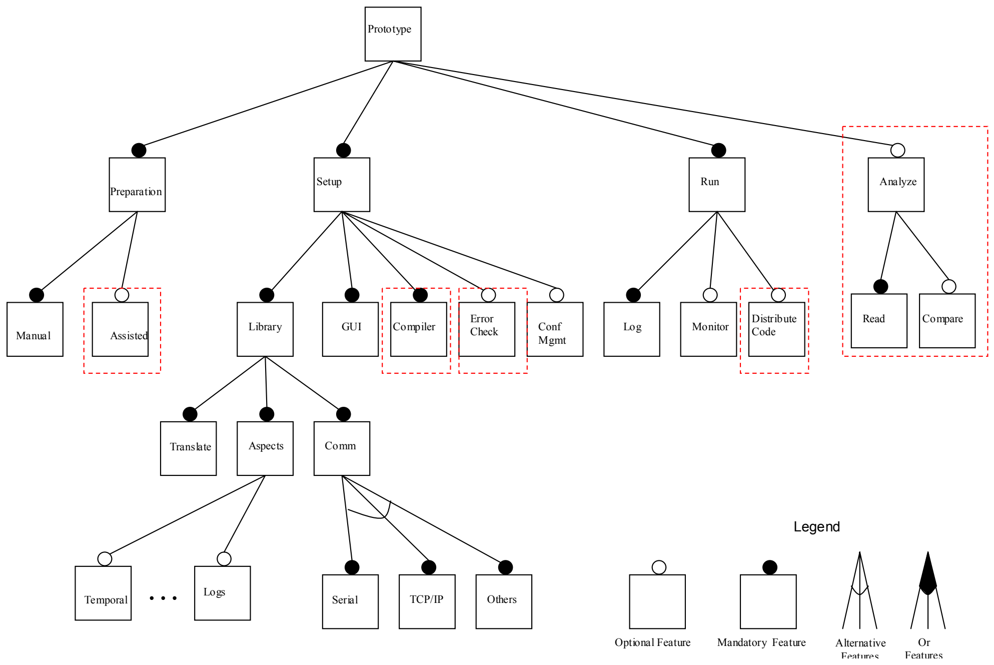


Figure 7 Feature Diagram for a Prototyping Environment

3.2.1. **Preperation.** In the preparation phase, Software experts create the interfaces and components necessary for compiling a wrapper. This is usually not a trivial task. Creating an interface to a module with only a serial output may take several weeks of effort, and may include creating wiring harnesses as well as code. The engineer charged with this task must first sort through the code and documentation (if available) to discover the format of the serial packets (Figure 8). Once the serial packet formats are understood, then a JAUS message that matches the packet data must be chosen. For the case of an ODIS command packet, a JAUS wrench command is chosen (Figure9).

Another part of the puzzle is to create a set of XML data structures to describe JAUS commands. A set of XML data structures will be associated with each translator component, one for each of the message types supported. Data generated during the translation will be passed through the wrapper in XML format.

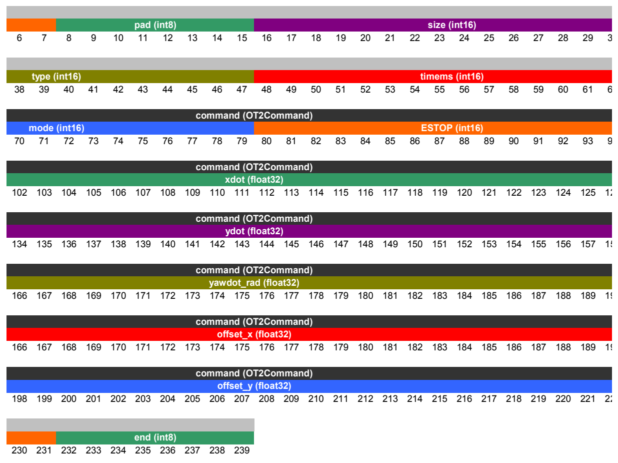


Figure 8 ODIS Serial Command Packet

Field #	Name	Type	Units	Interpretation
1	Presence Vector	Unsigned Short	N/A	See mapping table that follows.
2	Propulsive Linear Effort X	Short Integer	Percent	Scaled Integer Lower Limit = -100 Upper Limit = 100
3	Propulsive Linear Effort Y			
4	Propulsive Linear Effort Z			
5	Propulsive Rotational Effort X			
6	Propulsive Rotational Effort Y			
7	Propulsive Rotational Effort Z			
8	Resistive Linear Effort X	Byte	Percent	Scaled Integer Lower Limit = 0 Upper Limit = 100
9	Resistive Linear Effort Y			
10	Resistive Linear Effort Z			
11	Resistive Rotational Effort X			
12	Resistive Rotational Effort Y			
13	Resistive Rotational Effort Z			

Figure 9 JAUS Message 405h Set Wrench Effort

In addition to the translation component, a communication component must also be created for the opposite side of the wrapper. The communication component will be the wrapper interface to the prototyping environment. An open question is if converting the XML data structure to a JAUS wire format is necessary or desirable. Of course, multiple components are necessary since the data flow is directional; one each, translation and communication, component for read and write.

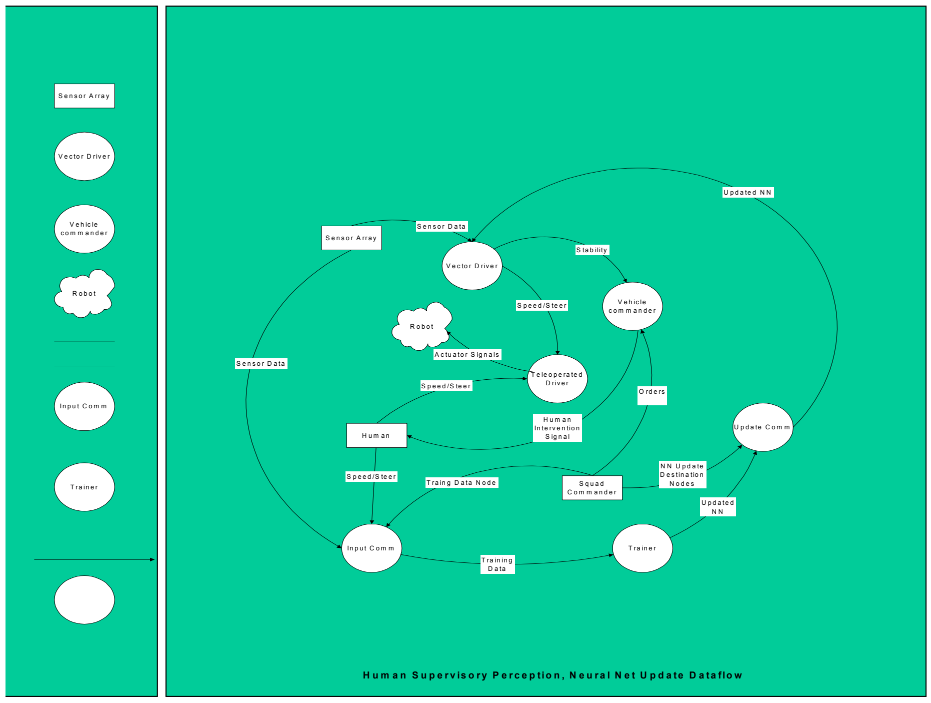


Figure 10 Model for Prototype GUI

Other components are necessary to make the prototyping environment useful. Logging, temporal logic, data throttle, and fault insertion are just a few. Programming elements, such as counters, loops and conditional gates are also necessary.

3.2.2. **Setup.** In the setup phase, the prototyping environment is turned over to a domain expert to construct a prototype system of interest (Figure 10). The engineer runs a Graphical User Interface (GUI) to create a dataflow model of a distributed system. Each node represents a separate computing element. When a node is opened, the screen changes and the wrapper programming environment is displayed.

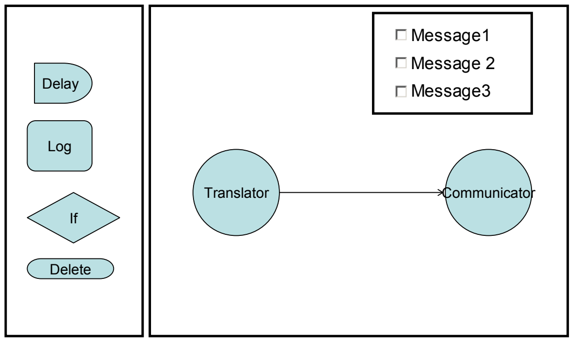


Figure 11a Wrapper Programmer, Initial Condition

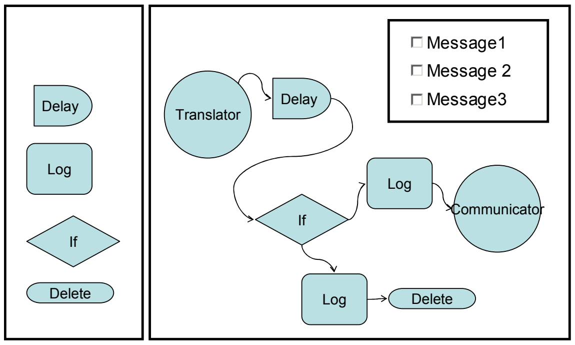


Figure 11b Wrapper Programmer, After Modification

Like the main programming environment, The wrapper programming screen will have components on the left and a work area to the right. In addition a list of allowed messages are displayed. If a message is checked, the environment will read the XML file for the message and display the data items in the checked messages. These will be used with context sensitive menus to configure components that are capable of making decisions based on message parameters. The results will be stored in a scratch file associated with the component graphical item.

The wrapper programming screen will start up with a translator and communicator component (Figure 11a). The user can accept this as it is, and complete any communicator setup, or the user can add additional aspects to the wrapper (Figure 11b). Since we are dealing with real time systems, timing is a concern. As we add additional aspect components, we slow down the system. We have to live with some overhead, and some of the overhead may be made up for by using faster communications than we expect in the completed system. In other cases, we may have to limit the aspects we install in the wrappers and possibly create two instances of a prototype with different aspects in each.

Additional standard utilities will be available such as storing the prototype at any time, retrieving a previous prototype, notes and cut/paste.

We now have everything needed in place to generate wrappers for each node. Generation will be discussed further in the next section.

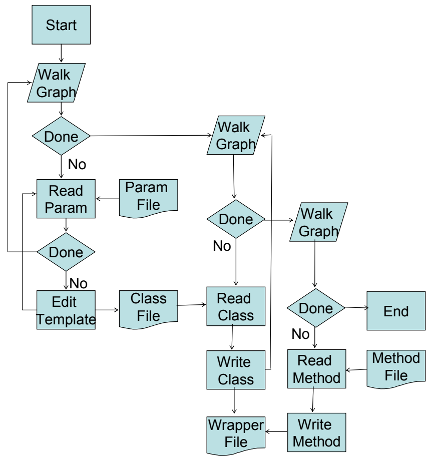


Figure 12 Wrapper Generator Flow Chart

4. Wrapper Generation

To generate the wrappers, requires that each node of the prototype be visited and the graphical program within the node be transformed into executable code (Figure 12). For each node we first edit each template with the appropriates parameter file. As the templates are completed, they are written to a file and a class is created. Finally, we walk the graph and call appropriate methods from each class on the graph, passing a reference to the XML version of the JAUS command. Each template includes in its instructions a file with machine readable instructions on how to invoke appropriate methods.

A grammar is being developed to formalize this task. A preliminary version of the grammar follows:

Wrapper → in-wrapper | out-wrapper
in-wrapper → communicator aspect-sequence translator | communicator translator
out-wrapper → translator aspect-sequence communicator | translator communicator
aspect-sequence → aspect-sequence aspect | aspect
aspect → log | delay | if | delete
log → **log-parameters log-template**
delay → **number delay-template**
communicator → **communicator-parameters communicator-template**
if → **if** exp **then** statement-sequence | **if** exp **then** statement-sequence **else** statement-sequence
delete → **end**
statement-sequence → statement-sequence **;** statement | statement
statement → edit-template | write-to-wrapper | call-next | end
edit-template → **if** parameter **then write** template
parameter → **log-parameters | communicator-parameters | number**
template → **log-template | delay-template | communicator-template**

5. Future Work

It should be clear if you have read this far, that this is a work in progress. I believe all the pieces are identified, and an architecture is nearing completion. But, there are a myriad of details that need to be completed to implement this work:

Formal rules are needed for composing translators, aspects and communicators.
The GUI has to be created.
A software base has to be integrated with the GUI.
The pop-ups and editing tools for the GUI need to be refined.
The current code generation scheme is simplistic and has can be refined, probably via several dimensions.
The run time monitor is but a dream, as is the analysis tools.

6. Conclusion

As mentioned in the introduction, software engineering will not come of age until automation penetrates the industry. Even today, the general perception of software engineers is that they are either coders or PC wizards. The Holy Grail for software engineers is generated code. The Holy Grail for any engineer is a tool that makes the job of those below easier and more economical.

---

[1] Luqi, V. Berzins, R.Yeh: A Prototyping Language for Real-Time Software, IEEE Transactions on Software Engineering, Vol 14, No10, Oct, 1988.
[2] P. Bourque et al., The Guide to the Software Engineering Body of Knowledge, IEEE Software, November/December 1999.
[3] C. Szyperski, Component Software: Beyond Object Oriented Programming. Addison-Wesley, 2002.
[4] K. Czarnecki and U. Eisenecker: Generative Programming, Addison Wesley, 2000.
[5] "Joint Architecture for Unmanned Systems." www.jauswg.org
[6] P. Young, V. Berzins, J. Ge, Luqi, Using an Object Oriented Model for Resolving Representational Differences between Heterogeneous Systems, SAC, Madrid, Spain, 2002.