Architecture and
Theory of Operation

SAVOY

FOR WINDOWS

*Web*Engine Platform Architecture

Revision History:

7/1/96	Original version for Release 2.1
9/30/97	Updated format for CD
12/97	Update for Release 3.0
	Capability-Based Access Control
	Arithmetic Logical Devices for Calculations
3/98	Update for Release 3.0
	WebEngine Partitions
	Capabilities Applied to Managing Device Name Space in Networks
8/98	Release 3.0E
	Embedded Ubiquitous Rule Processors
10/99	Release 3.2
	SPC Plug-in Client Framework
7/00	Rel 4.0C Console Application

Savoy Automation, Inc.
77 Salem End Lane
Framingham, MA 01702
http://www.savoysoft.com

© Copyright 1995-2003 by Savoy WebEngines, Inc.

All Rights Reserved.

The Savoy WebEngine System software and documentation contain information that is protected by copyright. No part of the software or the documentation may be reproduced, photocopied, translated, stored on a retrieval system, or transmitted, in whole or in part, without the express written consent of Savoy WebEngines, Inc.

Savoy WebEngines, Inc., reserves the right to make improvements or changes to any and all parts of its Savoy software, at any time, without any obligation to notify any person or entity of such changes. The software described in this manual is furnished under a license agreement. This software may be used or copied only in accordance with the terms of the agreement.

Savoy WebEngines, Inc., makes no representations or warranties with respect to the contents or use of this manual, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, **Savoy WebEngines, Inc.,** reserves the right to revise this publication and to make improvements or changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Companies, names, and data used in examples herein are fictitious unless otherwise noted.

Savoy WebEngines, Inc., has attempted to supply trademark information about all company names, products, and services mentioned in this manual. Other product names mentioned herein are for identification purposes only and are hereby acknowledged if they are trademarks and/or registered trademarks of their respective companies.

Savoy WebEngines is a trademark of Savoy WebEnginess, Inc.

Microsoft is a registered trademark of Microsoft Corp. Windows is a trademark of Microsoft Corp.

Architecture and Theory of Operations
Release 4.0

Document Number: 0796-21
First Edition: November 2005
Printed in the United States of America.

Preface

The information contained in this manual is provided to help you gain an understanding of the internal operation of the product and its components. Due to the pace of development in the automation industry, this manual may change frequently. We will attempt to keep it current with the state of the product and the direction of future enhancements, but cannot guarantee its accuracy at all times. Check our web site regularly for updates and contact a Customer Support representative at Savoy WebEngines for the latest information.

Other documentation available from Savoy WebEngines:

- *The Household Web*
- *Getting Started* booklet
- Type Manager series of booklets

These documents can be downloaded from the Savoy web site or ordered by phone from Savoy WebEngines.

Technical Support

Technical support is available to all customers via:

- E-mail support@savoysoft.com
- WWW site http://www.savoysoft.com
- Telephone 800-527-2853
- Fax 508-366-7845

Before contacting us, be sure to have available:

- the version number of your Savoy system
- the model name and number(s) of your supported automation devices
- a return phone, fax, or e-mail address

Table of Contents

Technical Support 3

TABLE OF FIGURES ____ 9

THE SAVOY WEBENGINE COMPUTING ENVIRONMENT ____ 11

Savoy WebEngine™ - a Networked Control System for Devices ____ 11

UBIQUITOUS RULE PROCESSING ____ 15

Device State 16
Assertions 16
Conditions 16
Rules 16
Device Name-scoping 17
Processing Element Categories 17

Component Organization 18

Client Applications 19

CONTRASTING SAVOY WITH OTHERS (LIKE SUN'S JINI, MICROSOFT'S HOME API, LONWORKS, ETC.) 21

RPCs, RMIs, and ORBs 21
Late Binding 22
The Savoy Architecture of Binding 22

DOMAINS AND TYPE MANAGERS ____ 25

Comparison to Object oriented languages 26
A Better Way of Building Procedures 27
Applications on top of the Savoy WebEngine 29
Savoy WebEngine as a System 29

THE WEBENGINE ____ 31

Network Configurations 31

Connections 31
Fault Recovery and Persistent Connections 32
Client/Server Protocols 32
Dial-up Modem connection 32

Rule Processor 32
Discussion of Computer Languages 33
The Rule Language 34
Name space of the Rule Language 36
Network Computing 37
Internal Functions 37
Many Views of a Rule 37
Passing Information Across Domains 38
Tunneling between Domains in the Savoy WebEngine 38
Templates 39
Rule Set Templates 39
Single Assertion Templates 40
Processing Rules in a Networked Environment 40
Performing Calculations in the Savoy WebEngine 42
Arithmetic Logical Devices 42
Random Numbers 42
Randomizing Sunset – an Example 43
Using the Numeric Constant Data Type 43
Java Scripts in WebEngine 45
Java Script API 46
Server commands 47
Invoking a Java Script 48
Arguments Passed to the Java Script 48
Simple Example 49
Java Scripts in the Embedded HTML Server 49
Capability-Based Access Control 51
Introduction 51
Capabilities in the Savoy WebEngine 51
Constructing Named Capabilities 52
Building a Hierarchy 55
Capabilities Applied to Enabling Layout access to a server 55
Customizing Capabilities 56
Examples: 56
Capabilities Applied to Device Name-Scoping in Peer Networks 57
Archival Storage 59
Time Ordered Event (toe) format 59
Reconstructing corrupted files 59
Saving Event Data 59
Rule Builder 59
Graphical Rule Builder 60
Basic Rule Builder 61
WebEngine Applications 61
WebEngine Partitions 63
Installing Multiple Partitions 63
Sharing Across Partitions 63
Network Applications of WebEngines 65
Devices and their Proxies 65

Assertions to Devices Propagate across the Network 65
Scalability and Name Scoping 65
Connecting Servers 66
Client Applications 66

THE CYBERHS DEVELOPMENT APPLICATION 67

Relationship to WebEngine 67
Layouts (Spatial view) 68
Local vs. networked 68
Strip Charts (Temporal view) 69
Device Context 69
Physical Properties 70
Style Properties 70
Animators 70
Sequences 70
Types 71

SAVOY CONSOLE APPLICATION 72

Operation 72
Description of Specific Plug-ins 74
Event Manager 74

NOTES 76

Table of Figures

Figure 1 Rule Compiler, Rule Processor, with Domain Manager[see below] Architecture 17
Figure 2 Downloaded Rule Compiler and Rule Processor Managing Specific Device Types 18
Figure 3 Server-based Compiler, Rule Processor, Specific Device 18
Figure 4: Methods and callbacks to the domain 26
Figure 5: Assertions to/from the Rule Processor 27
Figure 6: Peer-peer relationship of WebEngines 31
Figure 7: Naming physical devices 33
Figure 8: Naming logical objects 33
Figure 9: A network of rules 35
Figure 10: Example of effects driving causes in other rules 35
Figure 11: Multiple causes and effects 35
Figure 12: Executing rules with multiple cause phrases and effects 36
Figure 13: Rules cooperating across facility boundaries 41
Figure 14 Creating a new Java script called AnalogFilter 45
Figure 15 Graphical Editor in script mode 46
Figure 16: The Graphical Rule Builder (short form) 60
Figure 17: The Graphical Rule Builder (expanded form) 60
Figure 18: Rule Builder showing English sentence rules 61
Figure 19: Either RuleBuilder showing component modification 61
Figure 20: Server Application dialog box 62
Figure 22: Savoy WebEngine component overview 67
Figure 23: The CyberHs application’s desktop with a layout displayed 68
Figure 24: Strip Chart example 69
Figure 25 SPC login 72
Figure 26 SPC Setup dialog 73

The Savoy WebEngine Computing Environment

The Savoy WebEngine is a networked computer system that provides higher-level operating system services for managing objects in a real-world environment. Objects are cast into a common framework referred to as '**Devices**'. This framework provides the environment in which Devices can interact. Device interaction is controlled by the execution of a rule-based computing element, which permits device states to be compared, combined and controlled.

The WebEngine is designed to operate on networks of computers and is a combination of a client/server as well as peer-peer system that can be run all in a single machine, or across a network. Network operation can include local area Ethernet, dial-up modems, or Internet.

The WebEngine system can be applied to a broad base of applications that require the control and coordination of persistent Devices. Devices include a wide variety of categories such as communication devices (faxes, telephones, pagers, email), environmental devices (security systems, lighting, heating), computer-related devices, Internet sites, and more, within a home, a small business, a factory, or combinations of these connected across the Internet.

Savoy WebEngine™ - a Networked Control System for Devices

The Savoy WebEngine utilizes advanced concepts in object design and networking to provide a comprehensive, network based environment for the coordination and control of Device objects. Let's examine some of the characteristics of this system and how they compare with conventional approaches.

Characteristics of Devices:

Containment - a Device is a component that encapsulates both data and programs required to support the device's behavior. Devices have methods that can affect its state, and Devices have events generated from activity within the object.

Properties - data parameters that describe the state and activity of a device.

Persistence - once created, devices exist beyond the duration of any single program or computer activity.

Accessibility - Devices exist in a global network-wide name-space and are accessible from any location within that name-space.

Mobility - certain devices can move from one location to another.

Characteristics of Device Development:

Derivation (Inheritance) - Devices are derived from a root class that provides the minimal essential behavior required for the system (Assert, Evaluate TBD)

Composition - Devices can be combined to provide higher levels of behavior

Scalability - Devices can be added to the system dynamically in a manner that does not create disproportionately greater complexity.

Characteristics of the Device-based Networked Control System

Rule-based Control Logic - A rule is the fundamental computational element. It performs Evaluation functions of Devices, and Boolean logic tests on the states of devices which results in conditional changes in Device state.

Normalized Device Interaction - Base class of all Devices provides methods required to perform elementary rule calculations.

Ubiquitous Architecture - Scalable architecture permitting server functions to run in Internet servers, dedicated PCs, as a background task in an office PC, in an embedded Windows/CE processor, and in a Java VM. All implementations coordinate peer to peer using TCP/IP protocols.

Networked Name-Space - All interconnected servers exchange Device context, creating proxies for remote devices, and permitting rule operations to be performed on the network global set of devices.

Adaptability - The set of supported devices can be expanded by the development of additional Type Managers, which can be incrementally installed in a system without interruption of service. Type Managers can be designed to support practically any new Device, permitting device specific properties and assertions which are automatically applied to the rule logic.

Scalability - The concept of a scoped name-space is applied to device name propagation across the network of interconnected servers. By managing the scopes of name spaces, the device interaction is controlled, permitting very large systems to expand linearly.

As a networked control system for devices, the WebEngine represents a computing paradigm which can be compared to conventional computer systems by way of analogy:

Conventional system	**Savoy WebEngine**
Variables	*Device*
Variable Class (real, string..)	*Device Class (Domain: logical, X10…)*
Variable Type (short, long..)	*Device Type(LM465,…)*
Variables values (0,1 …)	*Device state(On, Off..)*
Variable operators (+,- ..)	*Device assertions(methods and events: On, Off..)*
Variable Name-scoping	*Device Name-scoping*
Language statement	*Rule*
Arithmetic - Logic computation	*Boolean computation of a Rule*
Language procedures	*Rulesets & Templates*
Code Module	*Network Server*
Linked Code Modules	*Peer Connected Servers*
Variable Scope (a Block {…})	*Scope Capability (subset of peer servers)*

Files	*Message Boxes*
File Data	*Messages*
File Names	*Message Box Names*
File System	*Message Repository*

Ubiquitous Rule Processing

It is generally assumed that all devices endowed with computers will somehow be able to communicate with each other – the industry has a sufficient number of proposed ways to do this [CEBUS, home network LANS, JINI, etc.]. Beyond ***how*** they will communicate, however, is the question of ***what*** they will communicate. That is, what is the logic, or set of algorithms, that determine overall behavior of a complex system of devices?

In some cases, a device may require specific services of another device. For example, an HVAC (Heating, Ventilation, and Air Conditioning) unit requires the specific services of a thermostat, a requirement designed into these particular devices. But suppose we wish to extend the 'design' of a device to provide an unforeseen interaction, like having your thermostat interact with a security system (reducing the temperature while away)? For this purpose, we need a way to extend the logic of a device.

In object-oriented computer languages (e.g., C++, Java), extending the design of an object is done by deriving another object from it and then adding methods to implement the extensions. However, if we added the extension as object methods, both participating devices would have to be 'designed' for the interaction; e.g., the security system would have to 'know' about thermostats. We have a stricter requirement: any device should be able to dynamically connect to any other device without any knowledge of their partners. To achieve this, we must impose some strict conformity to the extended methods.

Given this, a simple and compelling way to add logic to a device is to add standard methods that implement 'rules'. These methods include 'Assert', 'Evaluate', etc., and are discussed in detail later. Since all device extensions conform to these standard interfaces, any device could, in principle, fire rules in any other device. The determination of which rules to fire in which devices is not part of the device implementation, but rather encoded into a common rule data structure.

Many devices have 'implicit' rules – a thermostat has a rule that says 'when temperature is below the set-point, send message to turn on the heater'. With embedded computers inside the thermostat, we can enhance its behavior to also send a message to the security system (or whatever) – we call this an 'explicit' rule.

Adding explicit rules to a device can be implemented in one of several ways depending on the proximity of the processor to the device. If the processor is embedded within the device, rules can be compiled and downloaded into it. If the processor is a controller that connects to one or more devices, rules can similarly be loaded into the controller. Finally, if neither an embedded processor nor a programmed controller is appropriate, the rules can operate in a general-purpose server computer.

The advantage of a common rule structure across all of these implementations is (1) they can all compatibly interact with each other, (2) it is independent of the communication technique and can arguably utilize any communication facility available (particularly any standard that emerges), (3) does not require a common operating system nor implementation language to run in the processors, and (4) encodes the actual functionality of the interactions so that the design of devices themselves can remain generic.

We present here an overview of the fundamental concepts of how we model a device and how multiple devices interact through rules.

Device State

Devices are assumed to have a finite state. The total state of a device is the enumeration of all possible internal configurations or modes that the device can acquire. The entire system is therefore subject to the formal modeling techniques of finite-state systems.

Note: An important aspect of the WebEngine is the adherence to a finite-state model. Here, we concern ourselves with state transitions as opposed to (say) messages that cause events to happen.

Assertions

The transition from one state to another is the result of an 'assertion'. Assertions do not have direction: an assertion ***to*** a device that 'causes' a state transition is indistinguishable from an assertion ***from*** a device indicating the 'effect' of a transition.

Conditions

Closely related to an assertion is a **'condition'**. Conditions test whether a device is in a particular state or not and have a corresponding value of True or False.

Rules

By combining conditions and assertions, we can construct simple rules that describe how two or more devices should interact. The rule has three components; a list of conditions, and two lists of assertions -- a 'True list' and a 'False list'.

Rules have a simple state, either True or False, which is determined by the AND of all conditions in the condition list. Activity occurs when rules change state. When a rule goes from False to True, the True list is asserted, and when the rule goes from True to False, the False list is asserted. This rule construct has been shown to be simple, convenient, and sufficient with respect to describing most device interactions.

The logical OR-ing of conditions is accomplished by multiple rules, since all rules operate in parallel. Practice has shown that this approach is easy to understand and encourages good design principles thereby minimizing design errors.

Rule processors implemented in embedded devices have the constraint that the conditions of a rule must all refer to local state. Assertions, on the other hand, can be global to all scoped devices.

Rule processors implemented in server computers have no such constraint. In these systems, interconnected servers create proxy devices for all scoped remote devices. Consistency between these proxy devices and the remote real devices they represent is automatically maintained, permitting conditions to appear in rules (actually, it is the conditions of the proxy devices that are being tested).

Device Name-scoping

An important objective of this system is scalability and to achieve it we must eliminate the geometric growth in complexity associated with large numbers of devices. Name scoping permits the interaction of a practically unlimited number of devices without the encumbrance of geometric growth in complexity. It does so by permitting the construction of device-name hierarchies that requires that devices interact on an as-needed basis, in much the same way that computer languages scope variable names.

Devices interact according to whether they are in-scope or not. Scope [see section Device Name-scoping, page 17] determines the propagation of assertions across multiple interconnected processors. Devices that share a common scope are said to be 'in-scope' and fully interact according to prevailing rules.

Processing Element Categories

Rule processing is implemented in several different ways depending on the form of the processing element. To date, several distinct forms have emerged:

- Networked Computers having general I/O capabilities
- Embedded Attached Computers having compiler runtime capabilities
- Embedded Micro-processors having interpreted runtime system

From these definitions, there are three corresponding types of rule processors:

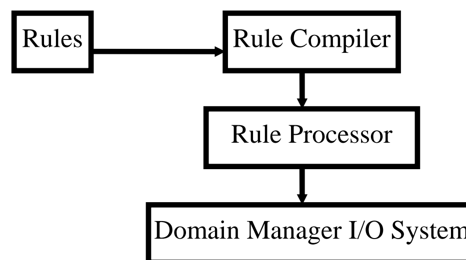


Figure 1 Rule Compiler, Rule Processor, with Domain Manager[see below] Architecture

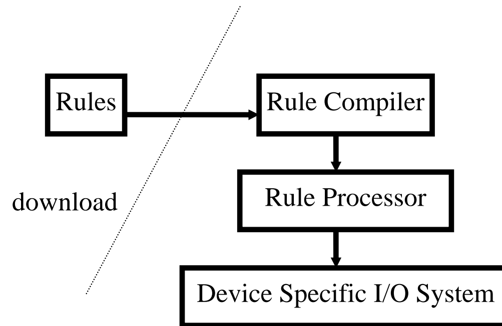


Figure 2 Downloaded Rule Compiler and Rule Processor Managing Specific Device Types

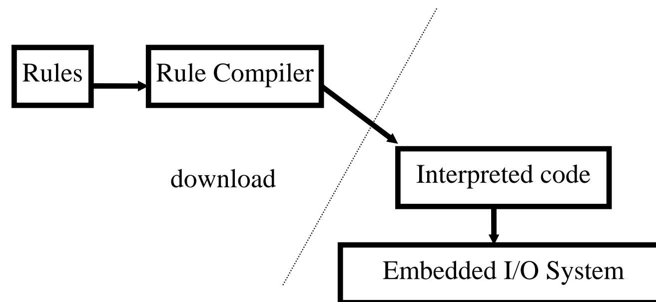
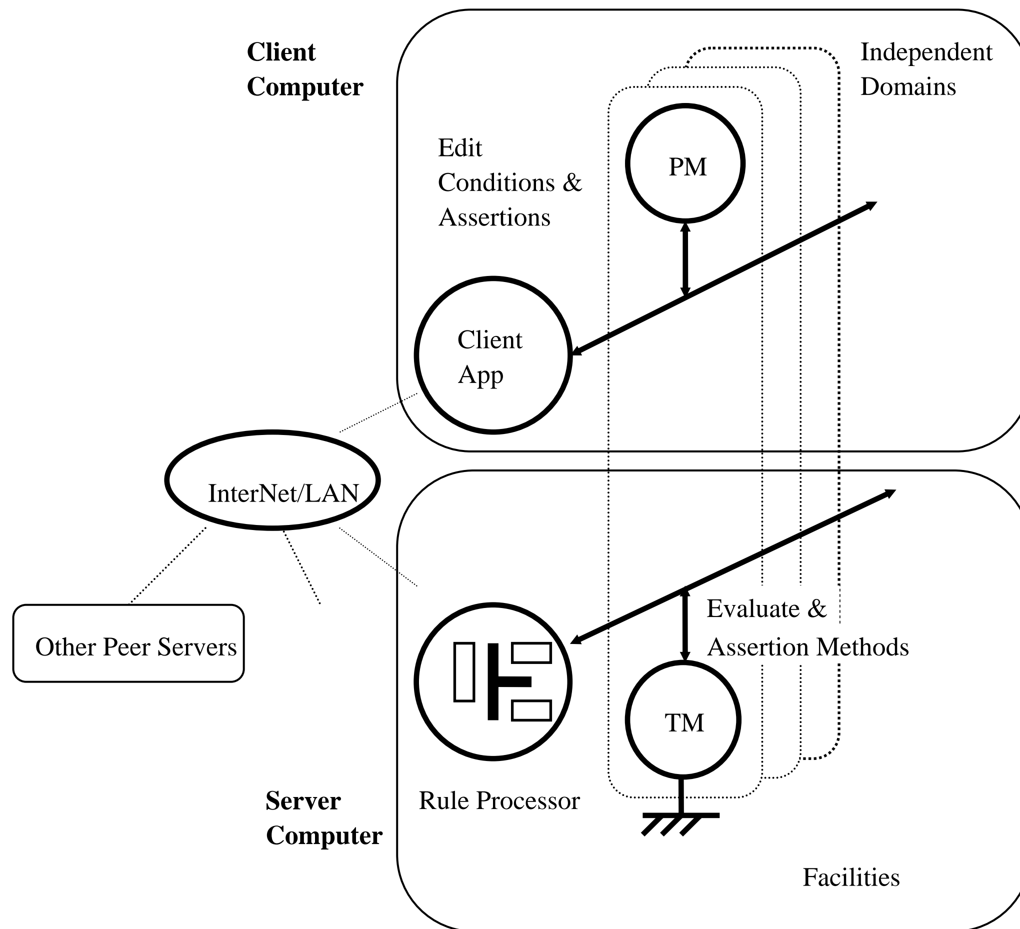


Figure 3 Server-based Compiler, Rule Processor, Specific Device

Component Organization

The Savoy WebEngine is organized into two principle types of components:

- computing elements that implement device behavior in the form of a server module, an attached embedded controller, or an embedded micro-processor within a device.
- a set of Client modules for development or viewing.



Multiple Servers can be linked together, functioning in a peer-peer mode. In general, the Server contains all of the logic to control devices, including the execution of a rule-based logic engine.

Client Applications

The Client applications include the capability for rendering and analyzing device behavior in both spatial and temporal views. Spatial views are physical layouts of an environment for the purposes of monitoring and control, while temporal views are a time-line history of causes and effects required to determine what happened and when. The Client applications also host application components that install and configure devices and an editor for creating rules that automate their operation.

For the WebEngine, the advantage of this network organization is twofold: first, the set of connected Servers can be running continuously, whereas the Client applications need only run when the user needs to access the system, and second, the Client applications can access the Server(s)

over networks. Actually, a Client application can connect to multiple Servers, and at the same time, a Server can be connected to multiple Clients. Savoy WebEngine Release 2.0 and beyond uses Internet TCP/IP protocols between the Server(s) and all Clients allowing access over the Internet.

Contrasting Savoy with Others *(like Sun's JINI, Microsoft's Home API, LonWorks, etc.)*

Let's say that you want the lights to turn on when you open the door.

First, if this is **all** you want to do, the problem reduces to that of communicating somehow with the two devices and causing the light to turn on whenever the door opens -- end of story.

If, however, there are a collection of doors and lights, and you want different things to happen under different circumstances, and you want to avoid complex programming, and so on, the problem becomes more complex.

The general problem is referred to as **binding** -- in this case binding the door to the light switch, and the question becomes **"What** do you bind and **when** do you bind it?". In the world of software, of course, the actual devices become encapsulated by software objects and the connection between the devices becomes some form of network, and so we cast the problem into one of binding software objects over a network.

RPCs, RMIs, and ORBs

Historically, this problem has been addressed by Remote Procedure Call (RPC) facilities in operating systems (Sun's Unix) which permitted a module on one machine to call a procedure on another. In 1987, Apollo Computer introduced a more sophisticated RPC using a network based 'broker' to aid in the binding. This technology was transferred to the Open Software Foundation (OSF) and eventually became the industry standard called CORBA (Common Object Request Broker Architecture).

With the more recent introduction of the Java language from Sun, there came along with it a new Virtual Machine (VM) environment and therefore a need to re-invent an RPC mechanism that they call Remote Method Invocation (RMI), all wrapped up in a concept called JINI. JINI is advantaged in its ability to move programs around the network, since all VMs run compatible code. This permits a 'thin client' approach to binding whereby the called machine 'serves' code that can run as an agent on the calling machine.

Late Binding

The above are all forms of 'late binding', which attempts to remove the function from hard core programming and implement it *later* during system configuration or in some more flexible programming environment like 'scripting'. But all of these techniques are used within the world of programming, and when used alone, some *programmer* has to determine that the light should go on when the door opens.

This begs the question, "How **late** can we bind these objects?". Can we remove the binding totally from the programming environment? The answer is yes, and there are numerous ways to do it; one approach is to transfer 'network variables' across the network, requiring that all devices be properly normalized so the output of one matches the input of another. LonWorks, BACNET and others take this approach. Another way is to have a database that determines what action is performed as the result of what events. But, who creates the data, how do the data elements relate to device capabilities? Further, if you add a new kind of device with entirely different capabilities, can the data approach accommodate it?

Three Approaches to Binding Network-based Objects:

- **Remote Procedure Calls**
  - Each procedure performs a unique function
  - Arbitrary parameters passed with each call
  - Requires programmatic binding
- **Network Variables**
  - Transmits real or integer data
  - Requires data normalization among objects
  - Limited device complexity
- **Changes of State**
  - Finite-state software objects encapsulate real devices.
  - None of the objects have programmatic knowledge of any other object.
  - There are no application programs (including the rule engine) with programmatic knowledge of any of the objects.
  - Objects alone are responsible for publishing an enumeration of their state and functions.
  - Objects alone evaluate what state they're currently in.
  - New object types can enter the system without interruption or modifications and immediately interact with other objects.

The Savoy Architecture of Binding

In Savoy, these data elements are components of **rules**, and these rules can be dynamically created, or modified by anyone at anytime -- without disturbing the operation of the system. To understand how the rule components are formed and used, let's create a metaphor for the system.

First, we cast all devices into a finite-state model, meaning that each has a finite number of properties and each property has a finite number of states (door is 'open' or 'closed'). Now, let's represent each device by (1) a collection of **lights**, and (2) a collection of **buttons**. There is a light associated with each state and when the device is in that state, the corresponding light turns on and all others turn off. Similarly, there is a button for each function the device can perform. We assume that all devices can fit this metaphor.

The function of a rule is to correlate lights and buttons. Note that our metaphor has removed any meaning associated with either a light or a button. To the rule engine, all lights and all buttons look the same. Furthermore, pushing a device's button does not require a procedural interface at all -- only a simple message. So, in this model, there is little point in connecting the device objects together in any programmatic way -- we need only send messages.

Finally, there is the question of how we wire up the lights and buttons to form rules; suppose all devices provide little descriptions of their lights and buttons -- small phrases that help a user choose among the lights and buttons. A rule editor uses these descriptions to give a user a simple way to author rules.

Let's bring our buttons and lights metaphor back to reality by replacing them with the little description tags mentioned before. These tags are strings that carry no meaning outside of the device object that creates them. The rule engine keeps these tags as components of rules and will send them back to the device for two purposes:

- to **evaluate** whether the device is currently in the state described by the tag, and
- to **assert**, requesting that the device perform a state-transition to the state described by the tag.

This simplified explanation of the Savoy architecture explains how it is that new device domains can be added to a system and immediately placed in rules. It further explains how Savoy is able to scale from the very simplest device to the most complex. It also suggests that the current emphasis on procedural interfaces to network based software components is probably over stated for many applications.

Domains and Type Managers

The network server is designed to manage a large and extensible set of Device types. The set of Device types having common physical characteristics are grouped into what is called a *domain*. Since many Devices represent real-world objects that can be controlled by the computer, the domain is typically the space of device types that a particular hardware interface can control.

The Server performs computation at two distinct levels: (1) within a domain, and (2) across one or more domains. Computation within a domain is usually limited to dealing with the I/O peculiarities of the interface and is encapsulated and hidden within a software component called a *Type Manager*—so named because it manages the set of device types defined within the domain. Computation across domains is performed by an entity called the Rule Processor.

The Rule Processor within the Server performs Boolean logic across domains in accordance with rules created by users. This is a simple, but powerful concept. For example, suppose one domain is the Internet WWW wherein a device type called ‘weather report’ periodically browses a known Internet site that contains weather forecasts. Suppose a second domain is an I/O system used to control water valves on a lawn sprinkler system. Then, with point-and-click simplicity, a user could modulate the scheduling of the lawn sprinklers in accordance with rain forecasts.

The interface between Type Managers and the Rule Processor involves what are called *assertions*. An assertion can be thought of as either a command to the domain or an event from the domain. Assertions appear in the Rule Builders used by the user and so they are typically quite simple and intuitive; they represent the total functionality of the device type.

Associated with an assertion is a *condition*. Conditions also appear in the Rule Builder as a cause phrase, and for many device types, the condition is identical to the assertion. A condition is a test of whether the device is in the state or not, and has a value of True or False. One of the main functions of a Type Manager is to perform these tests through a method called ‘Evaluate’.

Type Managers typically

- receive assertions from the Savoy WebEngine and perform whatever function is required within the domain
- monitor activity within the domain and translate this activity into assertions back to the WebEngine when appropriate
- perform evaluations of conditions and assertions whenever the Savoy WebEngine wants it to

Normally, a Type Manager will support multiple devices, usually through a single interface. These devices may be of different types—for example, a single I/O interface will support multiple device types (analog input, relay output, etc.) each of which may have many instances (4 Inputs, 3 Relays). Accordingly, the interface between the Server and a Type Manager includes methods to create and destroy instances of device types. The create method must return a unique value (an object handle) that is passed back and forth as a device identifier.

In addition to these, the WebEngine requires that the Type Managers provide editing dialogs for the user to edit both assertions and conditions. These are typically simple pop-up windows that prompt the user and validate the user’s input. Many editing dialogs present radio buttons so that users don’t

have to type anything, making type checking automatic. These editing dialogs run as separate components loaded by the WebEngine client application rather than the Server application (permitting the editing of rules from remote locations), and the WebEngine automatically handles the serialization and transmission of data between the client and server portions of a Type Manager.

In practice, then, two software components are required for each domain – one loaded by the Server and the other loaded by the WebEngine's main (client) application. This latter component is referred to as the *Property Manager*. Frequently, however, a single Property Manager can be used for multiple Type Managers if the Type Managers implement similar device types. For example, many security panels have similar capability and can all use a common Property Manager. Similarly, while there may be many interfaces to X-10, each requiring a unique Type Manager, they can all use the same Property Manager.

Since the editing (and therefore creation) of both conditions and assertions, as well as the execution of assertions, is all handled by the Type Manager, the WebEngine itself knows absolutely nothing about the domain, a concept termed *domain independence*.

Domain independence will permit the WebEngine to expand its scope to include many unforeseen domains, simply accomplished by additional Type Managers. These additional Type Managers can be incrementally added in the field without any changes to the WebEngine itself, user layouts, installed devices, or rules. Once a new Type Manager is installed, new devices can be added, and new/existing rules can use the new assertions.

Comparison to Object oriented languages

Further insight into the role of Assertions can be gained by comparison to similar concepts in object oriented languages such as C++.

In C++, calls to an object are termed *methods* and calls from the object are termed *callbacks*. In both cases, data is passed in the form of explicitly arguments whose types are defined by the language and checked by the compiler for correctness.

In Type Managers, we have Methods and Callbacks to the domain *interface*. To the domain itself, however, we've modified the approach slightly.

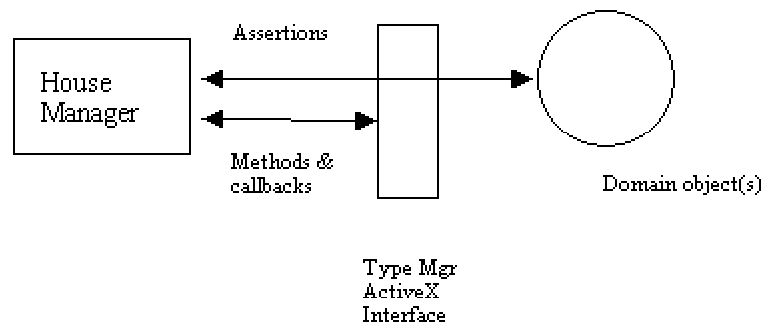


Figure 4: Methods and callbacks to the domain

First, we've named both methods and callbacks to/from domains *assertions*[1].

[1] The term *callback* is historic and somewhat unfortunate as well as condescending to the object, since the interface ought to be viewed as being symmetric—who's on top is merely a point of view. In our case, it might be viewed appropriate to name

Second, the data transferred across all assertions is not passed as individual arguments, but rather as a payload contained within a single string argument. This approach has two significant consequences:

1. A minor disadvantage of having all type checking performed at runtime and by the object itself rather than the compiler.
2. A major advantage of freeing the definition of the data from the C++ compiler in that a single interface can be used for a wide variety of data types, as is required for domain independence.

The advantage of (2) outweighs the disadvantage of (1), provided that all Type Managers implement adequate type checking of their assertions.

Further advantages of this approach become apparent when considering the task of building rule editors that must combine conditions and values of multiple domains. Here, the creation and editing of the data is securely performed by the Type Managers, and yet assembled and compiled by components in the WebEngine (Editors run on the client and rule compilers run on the server).

Finally, string data types are already serialized for communication in a distributed client/server configuration.

Symmetric assertions to/from domains provides a simple means of performing complex functions across the entire system. The user can regard the overall operation of the system as an asynchronous chaining of assertions to/from the rule processor:

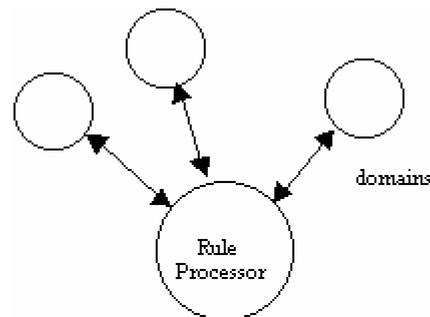


Figure 5: Assertions to/from the Rule Processor

It is important to realize the asynchronous character of this system: at any time, assertions can originate from within a domain, presumable due to some event occurring through an interface to a facility. Similarly, at any time, an assertion can originate from the Rule Processor, presumably due to either a timer event or from a rule triggered by an assertion from some domain. Rules simply perform Boolean logic on incoming assertions resulting in output assertions going back to the domains.

A Better Way of Building Procedures

A procedure is a sequence of operations having a specific order. Conventional approaches to encoding procedures involves a scripting language, like Basic, where each operation is specified by

---

Methods ‘Commands’, and to name CallBacks ‘Events’, but this asymmetry is troublesome—viz.: when you flip a light switch, have you created an Event or a Command? The function deserves symmetry, and so we’ve chosen ‘Assertion’.

a line of code and the system executes lines of code in sequence. This approach has two major problems: First, individual operations, especially those involving I/O to devices, can be time consuming and so to avoid the system from locking up during these operations, a form of preemptive multi-tasking must underlie the system to permit multiple script sequences to proceed concurrently. Second, frequently a single procedure may have sequences that could or should be performed concurrently, and to do this, the scripting language incorporates a multi-threading mechanism that permits concurrent threads to be created, synchronized, and so on. These conventional approaches are both complex and clumsy to use. Some scripting systems address these issues by a single scanning process that permits user code to test and set global flags that direct other sequences in the scanning loop to perform operations; these rudimentary techniques may work for simple applications, but don't deserve more than a mention when considering more complex applications. A better approach is required.

One better approach is to somehow encode the connections among a set of objects whereby each object is dynamically receiving messages (methods, assertions), operating on them, and then sending messages as a response. This, of course, is precisely what the WebEngine rule processor does, and the connection from the output of one object to the input of another is called a rule.

Rules can be designed to chain asynchronous assertions into procedures. Consider an application where the user wants to take a picture whenever a motion detector fires and then email the picture somewhere. Since the domains for motion detectors, cameras, and email all have asynchronous assertions, we can encode this procedure as a simple chaining of these assertions, as follows:

```
When motion | snap picture

When picture snapped | save to a file

When file saved | email to someone

When email sent | …
```

The advantage of this approach over procedural scripting becomes apparent in applications where long periods of time can elapse between assertions. The asynchronous design of the WebEngine permits a large number of these chained procedures to run concurrently.

In the above example, suppose the user wants to impose a 5 second delay after saving to the file. For this we use a standard `Timer`, called `Wait`:

```
When motion | snap picture

When picture snapped | save to a file

When file saved | Wait FiveSeconds

Wait Done | email to someone

When email sent | …
```

This shows that the notion of building procedures out of asynchronous assertions can be quite compelling. In other operating system environments, the function of the rule processor would be delegated to task switching, semaphore management, thread management, co-routining, or some equivalent form of parallel computing. One additional advantage of our approach is that it naturally spawns parallel threads: for example, if we wanted to log a message when the file is saved, we simply add a procedure of the form:

```
When file saved | Status The file was saved
```

The concept of parallel threads is a fallout of the fact that each individual rule statement is itself a parallel thread – so much so, that if one stacked up all of the statements of a all procedures and shuffled them like a deck of cards, the operation of the system would be unaffected.

In summary, the reader can regard assertions as asynchronous Methods and Callbacks to/from the domain object(s) with type independent data as the payload.

Applications on top of the Savoy WebEngine

The Savoy WebEngine client is based on components whose APIs are open, published and royalty-free. As such, they provide a rich and powerful base to build application specific client programs in Visual Basic, C++, or Java(soon). These application clients easily communicate with the WebEngine server(s) and can be designed to present specific user interfaces or control functions.

Savoy uses the client architecture to implement extensions to the system, such as a Voice Recognition module.

Savoy WebEngine as a System

In summary, the Savoy WebEngine is a computing environment for Devices that incorporates several new developments:

1. Device as a persistent object, and class hierarchy of device types derived from common base class
2. Device name space which spans networks of servers all operating in a concert, able to span embedded controllers to network servers, within a single site or across multiple sites.
3. Device interaction based on a rule-based computational element to provide the coordination and control.

The WebEngine

The Server component in the Savoy system is referred to as the WebEngine. This includes a Boolean Rule Processor designed to synthesize activity across classes of devices called *domains*.

The WebEngine normally runs continuously as a minimized server application on a Windows based computer, or in a more dedicated machine in a Windows 95/98/NT environment. Its principle function is to manage the devices local to the facility and execute the Rule Processor to automate the activity of these devices.

Network Configurations

While most environments will likely run in a single machine, the WebEngine is designed to cooperate with other WebEngines connected over networks. The WebEngines cooperate in such a way as to behave as a single entity, allowing for remote facilities to be managed collectively. Specifically, all WebEngines exchange cause and effect information allowing causes in one facility to have effects in another (see *Rule Processor* section on page 32).

The client/server relationship between the CyberHs application and WebEngine is supported by the Microsoft TCP/IP facility for routed and dial-up services, and Microsoft NetBEUI for LAN connections. Internet operation can be accomplished as provided that the WebEngine client knows the IP address of the server—easy if the server is running on a permanent site, more difficult for dial-up connections where IP addresses are dynamically assigned.

Connections

If more than one WebEngine is connected over a network, they function as peers all sharing a common device name-space. From a client's point-of-view, all interconnected WebEngines appear functionally as a single entity. The organization of the interconnect among peer level WebEngines is in the form of a parent-child tree as shown below. Servers can have any number of child servers, but only one parent.

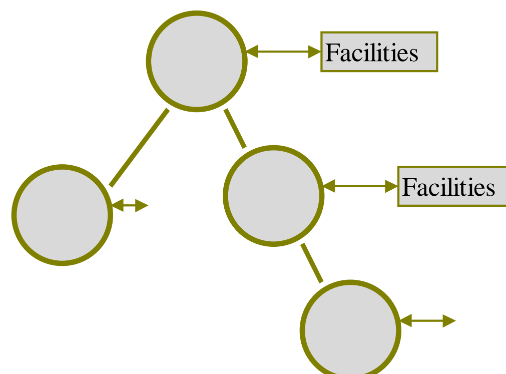


Figure 6: Peer-peer relationship of WebEngines

Each WebEngine is capable of managing facilities, and the system automatically routes messages up and down, permitting all WebEngines to see activities from all other WebEngines. Rules in any single WebEngine can be written for all devices across all WebEngines (see the *Name-scoping* section on page 42) without regard to their physical location, since the device name-space is shared across the system.

Fault Recovery and Persistent Connections

Approximately every minute, child WebEngines ‘ping’ their parent link with a null message to test the integrity of the connection. If this function fails, the child server goes into a slow polling mode, requesting service every 30 seconds or so, until the connection is reestablished. This allows for automatic reconnection in the event that a WebEngine anywhere in the tree is temporarily removed from service.

Client/Server Protocols

Both NetBEUI and TCP/IP are supported in Savoy WebEngine 2.0, the former recommended for LANs and the latter recommended for intranet and internet connections. Operation over the Internet requires that the client application know the IP address of the server, and the only way to do this currently is through permanent (non dial-up) connections.

A TCP/IP stack is not required for local or LAN operation; however, a WINSOCK.DLL is required. If TCP is absent, the WebEngine will log a warning and continue with NetBEUI.

TCP/IP is more efficient and for isolated LANs, users can set up an intranet using IP addresses 192.168.0.N where N is 0 to 255.

A single WebEngine can support multiple NetBEUI and TCP/IP clients at the same time.

Dial-up Modem connection

The Savoy WebEngine supports the Windows 95/98 Remote Access that allows for two or more Windows based computers to be connected temporarily over dial-up telephone lines. This connection is permitted under password control guaranteeing authorized access. This connection can be used for CyberHs application-WebEngine access, Server- Server access or both. Applications include the ability to access a facility from a remote portable computer as well as temporarily interconnecting remote facilities.

Rule Processor

The execution of the Rule Processor involves translating causes into effects according to one or more sets of rules.

Causes are the result of some environmental event including a user clicking on a button (in CyberHs application). Typically, a cause occurs when a motion detector sends a signal, or when an X-10 device is turned on or off.

The output of the Rule Processor is an effect, typically resulting in a command sent to an X-10 device, for example. The logic that determines the effect is described in rules that can range from extremely simple to quite complex.

Devices, such as wireless motion detectors or X-10 modules are referenced by a name, specified during the installation of the device. The rule processor does not distinguish between the various types of devices, and so, at the level of names, all devices are treated the same.

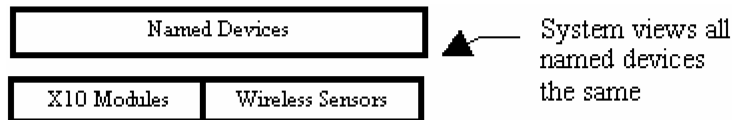


Figure 7: Naming physical devices

In addition to physical devices sharing a common name space, Type Managers have been developed for a wide variety of named objects including state variables that manage ‘scenes’ and ‘events’, Email messages, Web browsers, and more.

Figure 8: Naming logical objects

All objects can both generate causes as well as receive effects, which may in turn generate more causes. These devices allow for the construction of complex objects out of simple ones, which in turn greatly simplifies the rules that use them.

Discussion of Computer Languages

Most computer languages, like Basic, FORTRAN and C, are *procedural* languages meaning that they encode precise time ordered procedures for the computer to perform. Other languages are *descriptive* languages that encode a description of what to do without regard to how to do it. A subset of descriptive languages is *rule-based* languages that encode a set of logical rules to describe the systems behavior.

In general, procedural languages, even simple scripting languages, require a knowledge of the computer implementation, and frequently burden the user with complex issues of timing, synchronization, keeping track of multiple parallel processes, and so on. Rule based languages, however, bury all of that complexity into the language run time system, and provide the user with a far more elegant framework in which to encode certain types of applications.

The most popular rule based language is Prolog, which has been used in many artificial intelligence applications. There are many other rule-based languages that are integrated into applications, like Mail routing systems for example.

The Savoy WebEngine uses a rule-based language for encoding the logic for automation. Unlike Prolog, which is goal oriented, the WebEngine is event driven, and considerably simpler to learn.

The Rule Language

Everyone understands the simple relationship between cause and effect: namely, causes generate effects. This simple understanding is all that's required to create a set of rules that will automate the operation of the WebEngine. The syntax of the rule language takes the form

```
cause then effect;
```

which is to be interpreted as 'when the `cause` occurs perform the `effect`'.

A rule set is a sequence of these statements; they may appear in any order, and collectively they form the logic of the system.

An example of a rule set is:

```
cause1 then effect1;.

cause2 then effect2;.

cause3 then effect3;

. . .
```

Each of these causes or effects are called a *phrase*, typically a device-value pair, such as `Lamp On`. We define a *phrase list* to be a simple enumeration of phrases, depicted as:

```
Phrase1

Phrase2

Phrase3

...
```

A rule is comprised of three phrase Llsts. First is the list of causes, and the second and third are lists of effects, shown as:

```
PhraseC1 | PhraseT1

PhraseC2 | PhraseT2

PhraseC3 |----------------

PhraseC4 | PhraseF1

...
```

The operation of the rule processor is to assert the True phrase list whenever *all* cause phrases become true, and the False phrase list whenever *any* cause phrase becomes false. In this sense, cause phrases are AND'd together, while effect phrases are Or'd together.

An effect phrase is either terminated or connected to another cause phrase of another rule forming a network of rules.

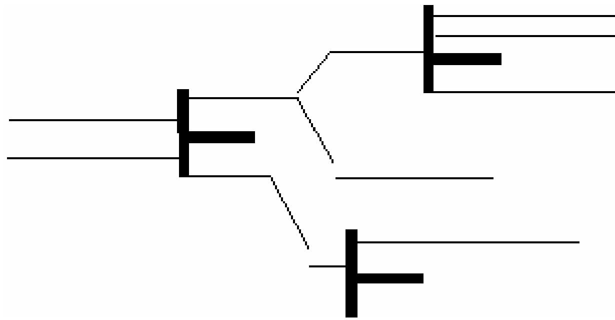


Figure 9: A network of rules

As an example, consider

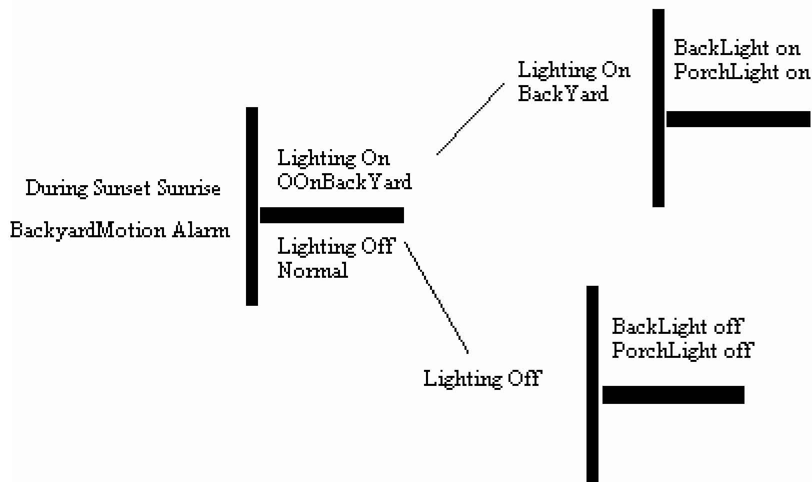


Figure 10: Example of effects driving causes in other rules

Each node in the rule network 'fires' whenever the AND of all cause phrases change from false to true, or from true to false. Suppose we have three phrases, named `A`, `B`, and `C`, forming the cause phrase list. Suppose further that the effect phrase lists are phrases `X` and `Y` for True and `Z` for False as shown below.

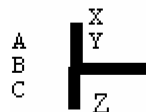


Figure 11: Multiple causes and effects

The operation of this rule is shown in Figure 12.

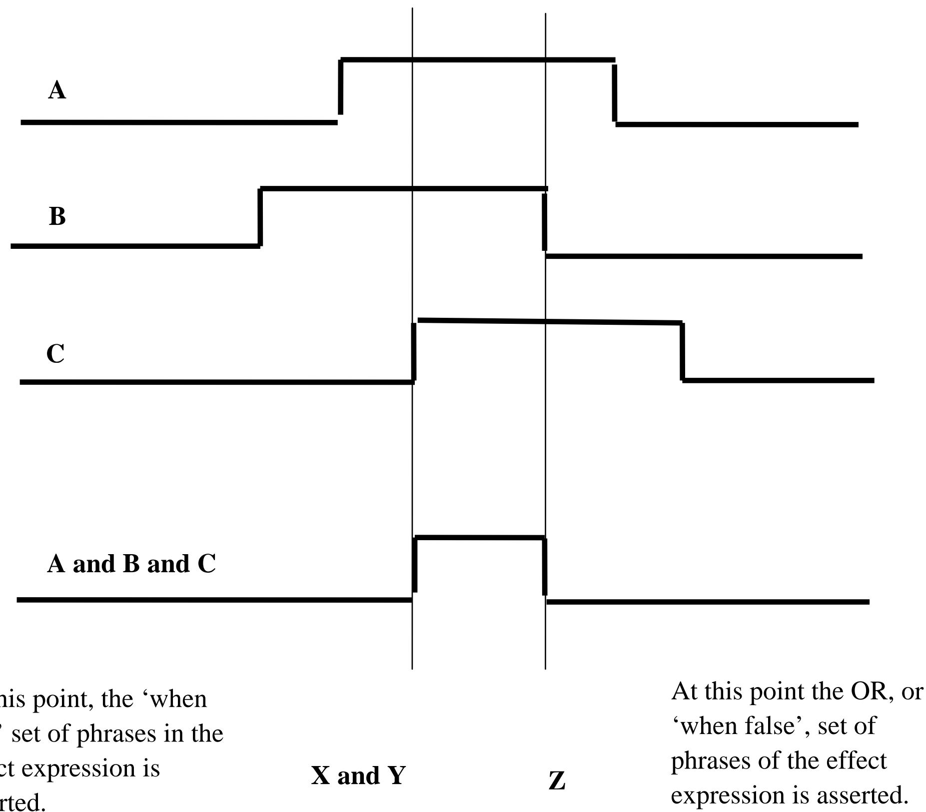


Figure 12: Executing rules with multiple cause phrases and effects

Name space of the Rule Language

Most generally, a phrase in a rule can be considered to be comprised of:

```
[ObjectName] | [Function] | [ObjectName Function] Argument(s)
```

Each component of a phrase is space delimited, so names used to identify the components cannot contain blanks.

`ObjectNames` is typically the name of a device, like `LampModule`. More generally, however, it can be a *logical* device created to hold state information, perform functions like timers, or abstract a collection of physical devices into a single logical entity, like `AllExteriorLights`. In conventional languages like Basic, these would be known as variables. Physical devices and named devices coexist in the same name space and are indistinguishable by the rule processor.

In addition, there are *virtual* devices that only exist within the rule sets—no physical or logical device by that name actually exists. These devices are dynamically replaced when a real or logical device is created. Conversely, if a real or logical device appears in a rule, and the device is deleted, a virtual device with that name is dynamically created.

The `Function` component of a phrase is typically `on`, `off`, or `alarm`, `clear`, etc. They indicate what the device is instructed to do, or what it has just reported (sensors).

Arguments are used to further qualify a function, for example `LampModule Dim 12`, meaning dim the lamp to level 12. Arguments are used to determine the value of the expression as in the case of the internal function `After` which compares the first argument to the time of day.

```
MotionDetector Alarm & After 18:00 | RingChimes & Lamp1 on;
```

A number of such internal functions exist to enrich the capabilities of the language.

Network Computing

Since the scoping of device names in the Savoy WebEngine can extend across peer servers running in parallel, the above approach works transparently when device objects are distributed across a network. Assertions to a remote device are automatically routed to the hosting machine, and assertions from the remote device are automatically routed back. Consequently, users can write distributed computing applications without regard to where device objects actually exist.

Internal Functions

A partial list of internal functions includes `After`, `At`, `Beep`, `Before`, `Date`, `During`, `Exec`, `Log`, `Status`, `Invoke`, `Expire`, and `Timer`. A complete list together with descriptions of their functions is available in the help documentation.

Many Views of a Rule

From our previous discussions of rules, we note the multiple views that this single entity can be presented. In the simplest case, it is a relationship between a cause and effect:

```
MotionDetector Alarm | Chimes On;
```

Or, it can be a procedure of assertions:

```
| Chimes On;
 Lamp1 On;
 Status blah blah;
 Relay3 Closed;
```

Or, it can be a complex procedure linking asynchronous assertions:

```
When motion | snap picture
When picture snapped | save to a file
When file saved | Wait FiveSeconds
Wait Done | email to someone
When email sent | RingChimes;
 Lamp1 on;
 Status blah blah;
 Relay3 Closed;
```

Finally, groups of these rules can be thought to form a network of rule logic performing very complex applications.

```
      Rule1 → Rule2
                   \
  → Rule3 …
```

Of course, these are all simply different views of the same thing. How any application is viewed is a personal choice and somewhat dependent on its complexity.

Passing Information Across Domains

In certain circumstances, it becomes necessary for information known within one domain to be passed to another. For example, an incoming phone call may have a `CallerID` value that is desired to be known by another domain, like a TV display.

To accomplish this in a general way, we attribute each device as having a set of property values, denoted by `DeviceName.Property`. For example, a device belonging to the Phone domain, called `IncomingCall`, may have a property called `CallerID` (jointly referred to as `IncomingCall.CallerID`), whose value at the time a call is received might be `Jones_Henry`. This value can be conveyed to any domain by simply stating the `device.property` name as an argument to an assertion.

For example:

```
IncomingCall Ring | Display IncomingCall.CallerID;
```

In general, Savoy's Rule Processor translates all string tokens having the form `a.b` (no spaces) into a reference to the property '`b`' of device '`a`.'

If device 'a' does not exist, the string '`a.b`' is passed as the value. If device '`a`' is created, all rule sets that include references to '`a`' are dynamically recompiled. If device '`a`' is deleted, all references to '`a`' are automatically converted back to '`a.b`.'

To encode an assertion that is comprised of a property, see the example of assertion templates accessing the properties of the Savoy WebEngine devices below:

Device Class	**Expression**	**Typical Value**
Any X-10 device	`Lamp1.state`	Dim12
Temperature	`Thermostat1.T`	72
Set Point	`Thermostat1.SP`	68
Mode	`Thermostat1.M`	H
All Security Panels	`Panel1.state`	Armed

Tunneling between Domains in the Savoy WebEngine

Tunneling is a technique that connects the output assertions of one domain directly to the input assertions of another domain, transferring free text arguments without modification or type checking.

The WebEngine Rule Processor performs tunneling between device A and device B by the rule:

```
A * | B *;
```

The asterisks indicate arbitrary arguments. There cannot be multiple assertions, and no 'else' assertions.

The Savoy WebEngine rule builders cannot create this rule since the '*'s violate type checking, and so we have created a simple template, called Tunnel, to create the rule. The template Tunnel has the form:

```
^1 * | ^2 *;
```

where ^1 and ^2 are device names passed in the Use statement that uses the template.

To use tunneling, create a rule with no cause assertions, and a single effect assertion of the form:

```
Use Tunnel DeviceA DeviceB;
```

This will cause all output assertions from device DeviceA to be sent to DeviceB.

Example:

Suppose a Visual Basic application, using the DDE Server Type Manager, wants to directly control a device, say an X-10 Lamp Module named Lamp1. Create a device of type DDEServer (Menu: Devices/New Device/Client Applications/DDE Server/String), called Lamp1Control. Then create the rule:

```
Use Tunnel Lamp1Control Lamp1;
```

Now, the Visual Basic application can directly create assertions of the form Lamp1 Dim7 and send it directly to the X-10 domain. Note that this is performed without type checking and so it is the responsibility of the VB application to guarantee the integrity of assertions it generates.

Templates

As in any computer application, value is gained through the ease of encoding higher forms of logic from simpler forms—procedures, structures, and object classes all lend themselves to this goal.

The Savoy WebEngine has the capability of using *templates* to accomplish this goal. A template is like a procedure, or macro in other languages—it encodes logic that can subsequently apply to multiple uses. In a template, names can be derived from arguments passed from a client rule set as in the example below.

Templates come in two forms: (1) an entire rule set, and (2) a single assertion.

Rule Set Templates

A template rule set named Momentary:

```
Timer ^1Tmr 0 | ^1 Off;

^1 On | Timer ^1Tmr ^2;
```

This rather cryptic rule set is designed to create a relay device which is momentary, meaning that it turns off a few seconds after it is turned on. The physical device, say Relay, is designed only to

turn on or off, but the application may require that it only momentarily turn on, lets say for 10 seconds, and then turn off. To do this, we simply state the rule

```
Use Momentary Relay 10
```

This statement transforms the conventional relay into a momentary relay. The user need never see the cryptic template rule set—they are contained in libraries. They can easily be developed by users, however, by simply observing the relationship between the arguments in the `Use` statement with the variables `^1, ^2, ... ^n` in the template rule set. Each `^n` is to converted to the nth argument when the rule set is compiled. Typically, however, the user need only understand the function of the template rule set and then encode the single statement shown above.

The momentary rule set can be applied to many types of devices in the same application.

Single Assertion Templates

An example of a single assertion template is `Property` which encodes the property of device `B` as an assertion to device `A`. The template is defined to be:

```
^1 ^2.^3;
```

An example of its use is:

```
(Conditions…) | Use Lamp1 Lamp2 state;
```

Users must design templates carefully, since their use is not type checked.

Processing Rules in a Networked Environment

A networked application of the system can range from a CyberHs application running on a laptop computer calling a WebEngine at home computer, to a complex of linked WebEngines and CyberHs applications at multiple sites. In all cases, the operation of the system follows a conceptually simple model.

In general, all named devices described above can be shared across the network of CyberHs applications and WebEngines. For example, a cause generated from a physical device at one site is sent to all other sites, allowing any site to generate effects. Consequently, a network of Rule Processors operate concurrently on all named devices, independent of where they are. Obviously, care must be taken when designing the rules at remote sites so they cooperate rather then interfere with each other. A simple example will illustrate how this works.

Suppose at site A we have a wireless motion detector called `Detector`, and at site B we have an X-10 device call `Chimes`, and we wish to have rules that allow for the chimes to ring at site B whenever motion occurs at site A. To implement this, we simply enter the rule `Detector | Chimes` at *either* site A or B—it doesn't matter which one. Whichever site is chosen, the rule processor will create a local soft device for the remote physical device. For example, if we choose site A, then on the left we have:

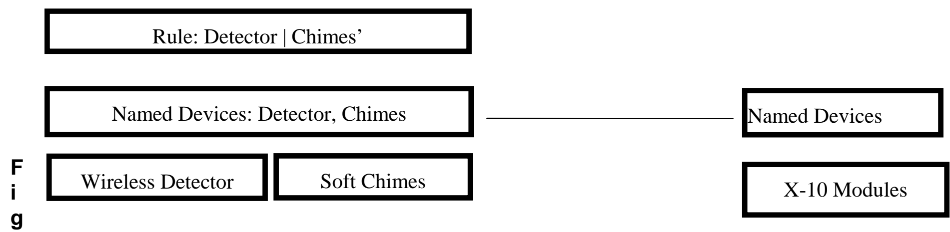


Figure 13: Rules cooperating across facility boundaries

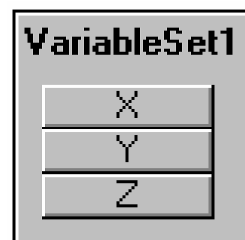
Consequently, at *each* site, we can describe rules for the entire system, regardless of where anything is, providing, of course, that we do not carelessly conflict with rules at other sites—one site turning something off while another turns it on, for example.

Performing Calculations in the Savoy WebEngine

Arithmetic Logical Devices

As of Release 2.5, the Savoy WebEngine has the capability to perform arithmetic calculations by a new logical device type called Arithmetic Variable Set (Devices Menu / New Device / Logical Devices/ Numeric Constants & Variables / Numeric Arithmetic Variable). This device computes assertions in the form of an arithmetic expression, similar to languages like FORTRAN and Basic.

Each device contains a set of up to 10 variables. Variables are properties of the device, and can be referenced as 'Device.Variable'. For example, an Arithmetic device might be called VariableSet1 and include three variables named X, Y, and Z. A rule for this device could be:




```
(cause) | VariableSet1 X= VariableSet1.Y + 3;
```

Any other device type which has numeric properties, like a thermostat, can be included in the arithmetic expression; for example,

```
(cause) | VariableSet1 Celsius=  ( Thermostat1.T - 32 ) * 9 / 5;
```

where Thermostat1 is a thermostat having a property T for current temperature. (Note that the property T for thermostats may be '72F' – non-numeric characters are ignored).

As in FORTRAN and Basic, items in parenthesis are computed first; parenthesis can be nested to any level. Operations include add (+), subtract(-), multiply(*) and divide(/) and are performed left-to-right. All numbers are assumed to be integers.

Random Numbers

A special Arithmetic device, called SysVars (Sytem Variable), can be created for computing generic values such as sunrise, sunset, and random numbers. To do this, create the device (Menu New Device / Logical Devices/ Numeric Constants and Variables / Std Time Var Set).

To reference a random number, use SysVar1.random (will be an integer = 0..99).

To cycle the number generator to the next value, assert "SysVar1 random".

Example:

```
(some condition… )      | SysVar1 random;
                          VarSet X = SysVar1.random * 3 / 6 …;
```

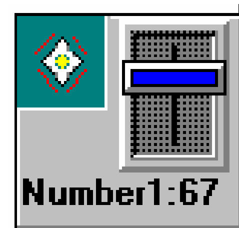
Randomizing Sunset – an Example

Suppose it is desired to randomize the timing of a light turning on by Sunset +/- 20 minutes. To do this, create two Variable set devices: one of type SysVar as above, another of type Time, named TimeSet. Then create a rule:

```
At 00:01:00        | SysVar1 random;
                   | VarSet X = SysVar1.random * 3 / 6 …;
                   | TimeSet T1= X.sunset + ( ( ( X.random - 50 ) * 60 / 50 ) * 20 );

At TimeSet.T1      | Lights On;
```

Using the Numeric Constant Data Type



The Numeric string device (Devices Menu / New Device / Logical Devices/ Numeric Constants & Variables / Numeric Constant) permits a user to input numbers 0 through 100 using a slider control. This device can accept an '*' as a rule condition indicating any change in value. Its current setting can be referenced as Number1.state, and so we can combine this device with an Arithmetic device using the following rule:

```
Number1 * | VariableSet1 Z=((65+ ( Number1.state / 10 ) ) - 32 ) *5/9;
```

which will translate the range of the slider into Celsius over a range of 65 to 75 degrees Fahrenheit. Suppose we want to have this value adjust a thermostat; simply add the assertion:

```
… Thermostat1 SH= VariableSet1.Z;
```

Java Scripts in WebEngine

Activity in the WebEngine is normally controlled by the operation of a rule engine that receives assertions from one device and sends assertions to one or more other devices. Since devices are all cast into a finite-state model, and since each assertion represents a defined state of a device, the rule engine can coordinate complex inter-device activity without any programmatic interface between devices. Each device can be designed as an independent entity without any knowledge of any of the other devices.

Occasionally, assertions require more complex processing than that afforded by simple state transitions. There may be the requirement to convert units (say Centigrade to Fahrenheit), or perform analysis of analog values, and so on. In principle, many of these requirements could be done within the Domain of the device, but there may be value in providing users with more flexible control over the calculations.

To meet these requirements, Release 4.0 of the WebEngine will include the Java scripts to augment the operation of the rule engine.

The concept is that the Savoy rule engine and an embedded Java script engine sit side-by-side within the server. Rulesets are typed according to whether they are (a) a set of rules or (b) a Java script and are thereby directed to the appropriate engine when loaded.

Users can create Java scripts in a similar way that rule sets are created. When a new ruleset is created, a user can simply check the box labeled 'Java script'. The Graphical Rule Editor automatically adjusts according to the ruleset loaded permitting the editing of scripts by normal text editing.

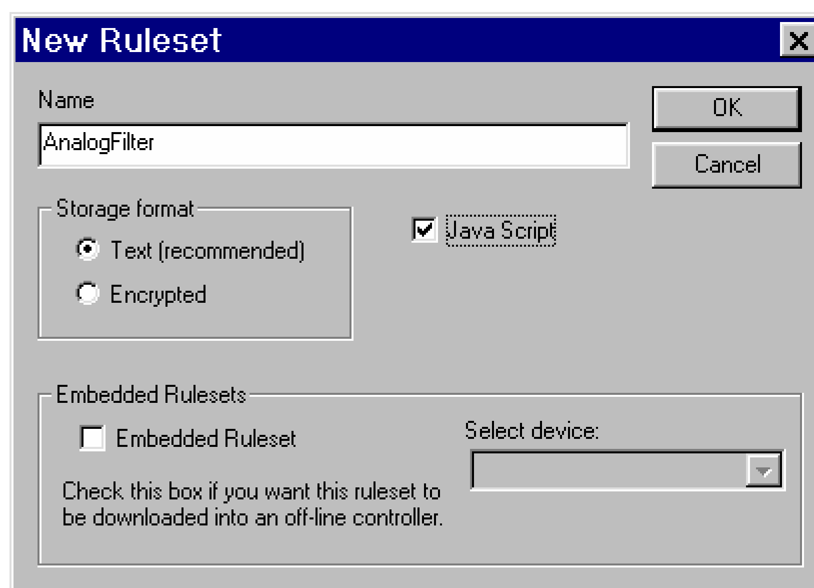


Figure 14 Creating a new Java script called AnalogFilter

When this script is loaded into the Graphical Rule Editor, the editor automatically adjusts to permit text editing.

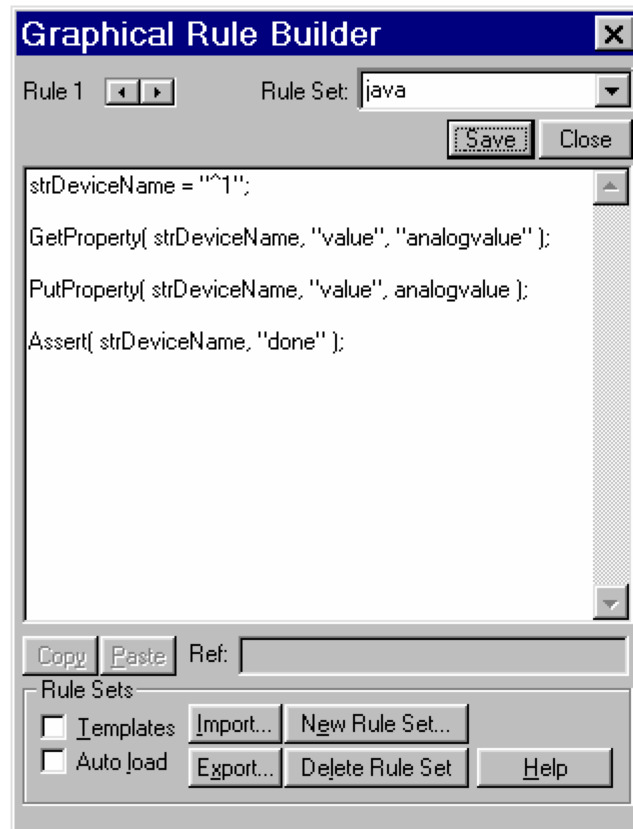


Figure 15 Graphical Editor in script mode

Java Script API

In addition to the standard runtime system provided by the Java script engine, a number of procedures are available to facilitate communication with the WebEngine environment. These include:

```
GetProperty( strDevicename, strPropertyname, strVariablename );
```

Retrieves a property of a device and stores it in a variable.

```
PutProperty( strDevicename, strPropertyname, strValue );
```

Sets a property of a device to a value.

```
Assert( strDevicename, strValue );
```

Asserts a value to a device.

```
Bitwise( operator, arg1, arg2, result );
```

Parameters

operator

A string specifying the bitwise boolean operation to perform. Must be one of “AND”, “XOR”, or “OR”.

arg1,arg2

A signed, decimal integer. (*Note: java script values are passed as a string)*

result

The name of the variable to which the result will be stored.

Remarks

Performs the bitwise boolean operation, as specified by the *operator* parameter, on the values *arg1* and *agr2* and writes the result to the variable specified by *result*.

Example:

arg1=14;

arg2=5;

result=0;

Bitwise(“AND”,arg1,arg2,”result”);

//result now has the value: 4

Server commands

The **Assert(strDevicename,strValue)** procedure includes assertions to the Server which will perform server functions including:

```
Assert( "Server", "Status " + strMessage );
```

Lists strMessage to the status window.

```
Assert( "Server", "Log " + strMessage );
```

Logs strMessage to the archive file.

```
Assert( "Server", "Invoke " + strRuleset );
```

Invokes a ruleset or Java script.

(more…)

Invoking a Java Script

Java scripts are invoked in the same way that rulesets are invoked. If the script is set to 'Autoload', it is run when the server is started or when the script is recompiled. Normally, however, a script will be explicitly invoked by a rule:

```
Cause(s) … | Server Invoke JavaScriptName( Arg1, Arg2, … ArgN
);
```

The execution of the script must run to completion before rule processing can continue. Arguments to the Java procedure can include Device.Property references; however, they must be surrounded by spaces in order for proper substitution to work. So, a space before the argument and another before the comma will permit Device.Property references.

Arguments Passed to the Java Script

The sequence of arguments **(Arg1,Arg2,…ArgN)** are referenced within the script by setting variables to values referred to as ^1, ^2, …^N (same as Template syntax). Typically, the Java script would include statements of the form:

```
Arg1 = "^1";
Arg2 = "^2";
Arg2 = "^3";
```

Prior to executing the script, the tokens ^N will be replaced by the string tokens provided in the Invoke assertion. In this way, a single script can be applied to multiple circumstances by appropriately setting the argument values.

Simple Example

Let's imagine a device that receives an analog value that requires special processing -- perhaps some filtering operation. We design the Domain software so that new analog values are put into the property named 'value' after which the device asserts 'newdata'. In response to this assertion, a rule fires that causes a Java script to run to perform the calculations. Upon completion, the script asserts 'done' to permit rules to process the calculation results. Since there may be many devices subject to this requirement, we create a single script and pass the device name as an argument.

One of many rules that invokes the script is

```
Device newdata | Server Invoke AnalogFilter(Device);
```

The script is

```
strDeviceName = "^1";
GetProperty( strDeviceName, "value", "analogvalue" );
[perform calculation on variable 'analogvalue']
PutProperty( strDeviceName, "value", analogvalue );
Assert( strDeviceName, "done" );
```

The rule that fires after the script completes is

```
Device done | …;
```

Note that in GetProperty(), analogvalue was quoted because the argument requires the variable name, not its value. PutProperty(), however, requires its value, and is therefore not quoted.

Java Scripts in the Embedded HTML Server

Release 4.0 of Savoy's Server also includes an embedded HTML server that is viewable from most standard Internet browsers. This server supports Active Server Pages (ASP) and Java Scripting, permitting users to customize their THML presentations. The Java scripts can also utilize the previously described API.

Several additional procedures are provided for scripting in an ASP, including:

```
AspGetString( strSection, strEntry, strParameter, strFile );
```

```
AspGetObjectName( strNumber, strType );
```

where strType = [device | domain | commport | ruleset]

```
AspSetVar( strVariableName, strVarType );
```

where strVarType = [DeviceCount | DomainCount | EventCount | RulesetCount]

Refer to distributed ASP files regarding usage of these calls.

Capability-Based Access Control

Introduction

The purpose of access control is to provide different levels of authorization for different classes of clients. For example, some may require access to create and modify rules, others may not.

In computer applications, the most common form of access control is termed 'authority based' whereby each resource in the system maintains authority over who has what access rights, typically accomplished by maintaining an 'access control list' attached to the resource. Common file systems are implemented this way, as are logon procedures.

An alternative to 'authority based' systems is what is termed 'Capability-Based' access control whereby the client is granted a ***capability*** that describes a set of access rights. Capability systems are more complex because provisions must be implemented to prevent unauthorized creation of capabilities. However, they have considerable advantage over authority based systems in terms of their flexibility in the assignment of rights to clients.

An authority based system is analogous to placing security guards at every door in a building, whereas a Capability-Based system is analogous to providing all clients with their own set of keys to unlock the doors.

The flexibility afforded by Capability-Based systems stems in part from the notion that a capability can both contain other capabilities as well as be contained by some other capability. Following our analogy, additional keys can be found inside rooms. This implies a natural hierarchy of capabilities – the higher in the hierarchy, the greater the rights. Capabilities need not be strictly hierarchical, however, but rather, can include an arbitrary assembly of lower level capabilities.

Capabilities in the Savoy WebEngine

Because of this considerable flexibility, a Capability-Based access control has been implemented in the Savoy WebEngine to provide selective access to many important features including device modification, rule editing, and more.

To explain the behavior of the Savoy WebEngine Capability system, we define a few terms:

A **resource** is a system component such as a device, a rule set, a client Layout, and so on. Resources are said to have **resource capabilities**. For example, the resource capabilities of devices include the ability to view, create, modify, and delete. Resource capabilities of a Layout include setting properties, design mode operations, viewing the event log, and so on. In order for a client (like a Layout) to perform some function on a resource, it must first possess the capability to do so. Individual resource capabilities are too numerous to deal with and so we construct a higher level capability called a **named capability**.

A named capability is an assembly of resource capabilities. A named capability in the WebEngine has a name, like 'EditRules', 'ModifyDevices', 'User', or 'Programmer' with quite arbitrary meaning depending on the specific capabilities assigned. For example, EditRules is an appropriate name for a capability that allows access to a rule editor. However, 'User' may be the total capabilities given to a particular client of the system, comprised of sets of lower level capabilities.

The Savoy WebEngine uses capabilities in two distinct ways: one to control access to the WebEngine from client Layouts, and second, to control which device names are shared across peer to peer networks of servers. While both applications use a common mechanism, the capabilities are independent of each other and selectable by radio buttons in the Capabilities dialog (below).

To control client access from a Layout, a single named capability is given to each client Layout, specified in Layout Properties dialog. When opening a new Layout, or modifying the properties of an existing one, a client is free to enter the name of a capability, providing, of course that the required password is correct. When the server connection is established, the named capability will be translated into specific resource level capabilities automatically.

Constructing Named Capabilities

The construction of a capability in the Savoy WebEngine may best be explained by the use of the dialog box that creates and maintains them. The following dialog is presented by the WebEngine [File Menu, Access…]:

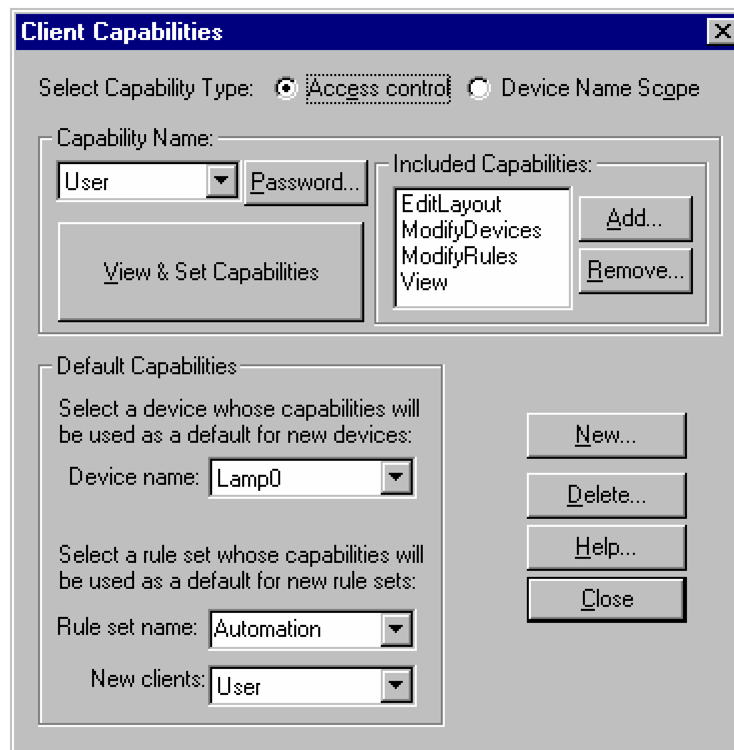


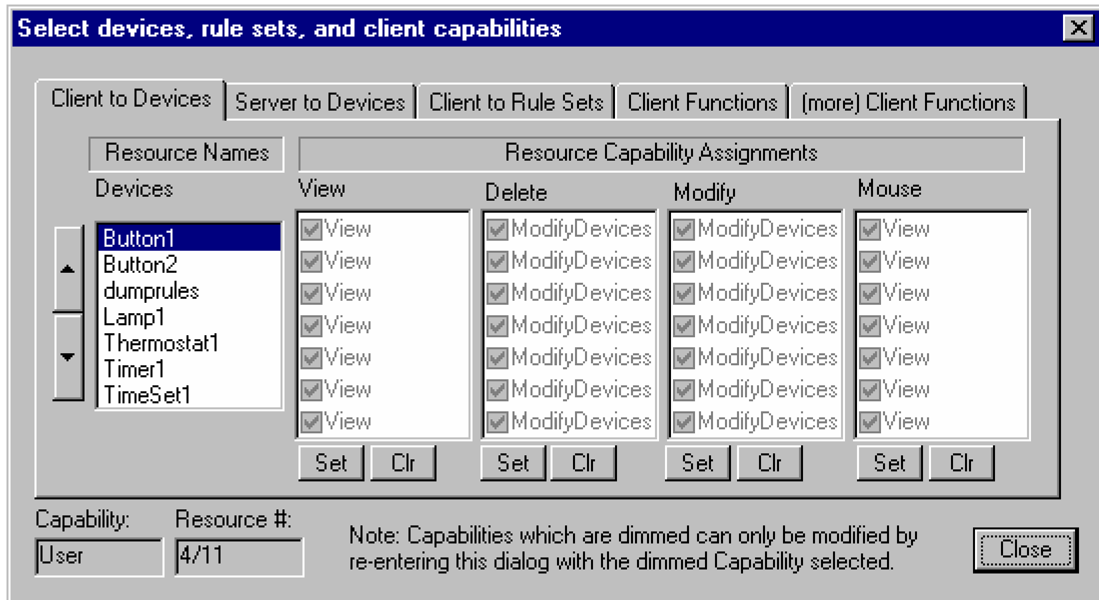
Fig 1. WebEngine's Capability dialog box

The following discussion will apply to the Capability Type 'Access Control'. The 'Device Name Scope' will be described in a later section.

Capabilities are listed in a pull-down list box and buttons are provided to create new capabilities [New…] as well as delete them [Delete…]. Each capability has an associated password without which the client cannot even show the capability in the dialog. If the password of a capability is blank, it is ignored.

The lower list box lists all included capabilities. Double clicking on any of these will cause the selected capability to be displayed.

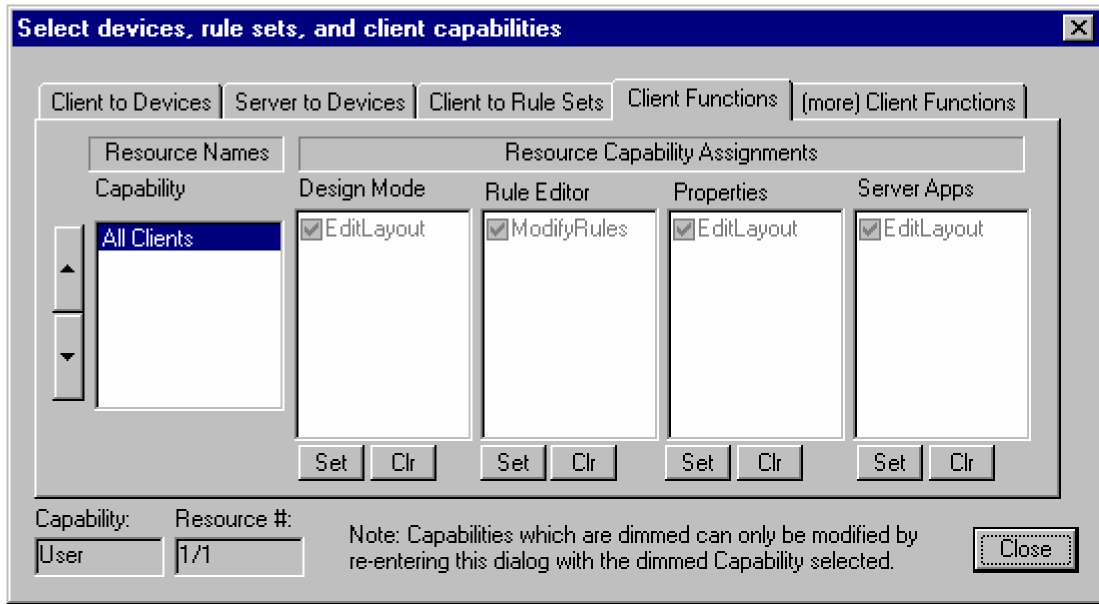
To set or modify a named capability, click "View and Set Capabilities…" and the following dialog will appear:



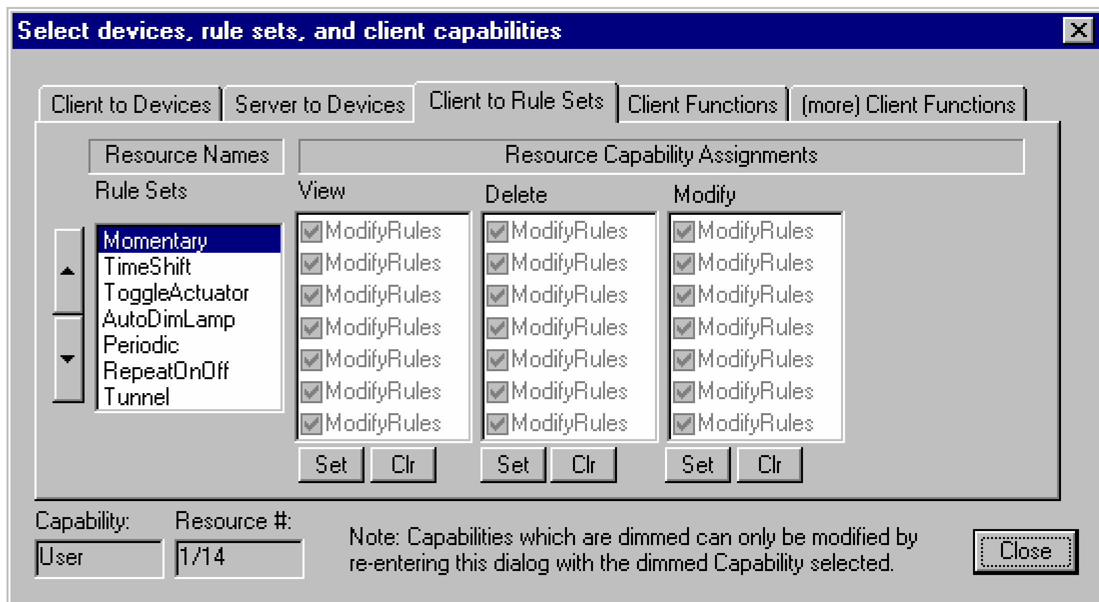
Each tab selects a category of resource capabilities, listing the resource (e.g., Devices) on the left, and the named capability (or "none") that is currently assigned to the resource capability. If the current assignment is a capability other than the current selection (shown bottom left of the dialog), then the entry is grayed. You cannot reassign a capability without unassigning it first.

By selecting All/Clear All buttons for all list boxes will attempt to assign or clear entries. You can also individually make assignments by setting/clearing individual checkboxes.

While there are lists devices and rule sets, there is only a single entry for Client resource capabilities. For convenience, the capability assignments are listed for all included capabilities so that the user can look down the list to see whether the resource capability is implicitly assign through inclusion of another capability, as shown below.



For example, note that capability "User" has no explicit capabilities for Client functions, and yet it will be able to perform these functions because it includes other capabilities that do.



Note that for both devices and rulesets, there can be only a single named capability assign to each resource capability. For example, suppose access to the Modify resource capability has previously been assigned to say Appliance1 in the capability 'ModifyDevices'. Later, if the capability 'User' is selected, then when the Devices tab is selected and the devices are listed, the checkbox associated with Appliance1 will be grayed (neither checked nor unchecked), indicating that the Modify capability is currently assigned to another capability. In order to reassign this device, the user must

first display the capability “ModifyDevices” and unassign it. Then, return to “User” to reassign it, providing appropriate passwords if required.

Building a Hierarchy

The lower portion of the dialog in Figure 1 shows a list box entitled ‘Included Capabilities’ which embodies a powerful concept. Any capability can automatically include another capability by simply adding [Add…] it to this list. As shown, the ‘User’ capability is actually a composite of capabilities including ‘DeleteDevices’, ‘EditRules’, ‘ModDevices’, and ‘ViewDevices’ (plus any capabilities that are, in turn, included in these). Of course, each of these capabilities can have access to rule sets and devices on an individual basis.

A recommended approach to providing secure access to a Savoy WebEngine system begins by first dividing Devices and Rulesets into groups of varying security levels. For example, consider all critical devices that relate to a security system (say) in one group, devices for casual lighting control in another group, and so on. For each group of Devices and for each group of Ruelsets, create a capability and assign Modify rights to it, checking each device belonging to the group.

At the next level, create capabilities that combine capabilities of Devices and Rulesets. For example, there may be a Device group that goes along with a Ruleset group, and so it makes sense to have a single capability that ‘includes’ both lower ones.

Then, create a capability for types of applications: one for installation that includes all capabilities, others for casual use that includes only View capability, and so on.

All critical capabilities should be assigned a unique password known only to the creator/owner of the capability.

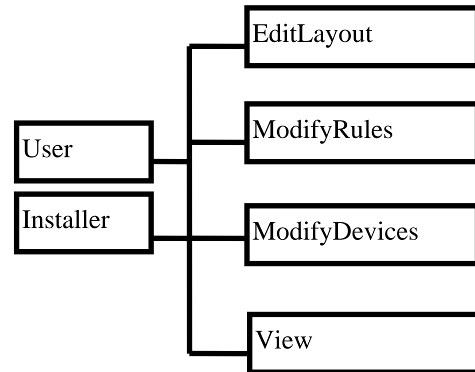
Capabilities Applied to Enabling Layout access to a server

The final step is to create / modify Layouts by assigning a single capability to them along with the password. Note that a Layouts capability can include or exclude the ability to modify its properties, and so if excluded, the Layout is sealed forever – or, until the owner of the capability modifies it at the WebEngine.

Layouts in the Savoy WebEngine have the option of retaining the password, or requiring it every time its opened. In either case, Layouts that deny access to their own properties are bound to a particular WebEngine with a set of capabilities that cannot be modified, even by the owner of the Layout. For example, users may make available a view-only Layout for others to use without thereby giving away the ability to affect their home server.

For compatibility with Layouts created prior to Release 2.5, the Savoy WebEngine assigns a default capability named “User”. Furthermore, Releases 2.5 and later come with the WebEngine having the following Capability hierarchy:

EditLayout includes the capability to modify Properties so that the named capability of Layouts created prior to Rel 2.5 can be reassigned.

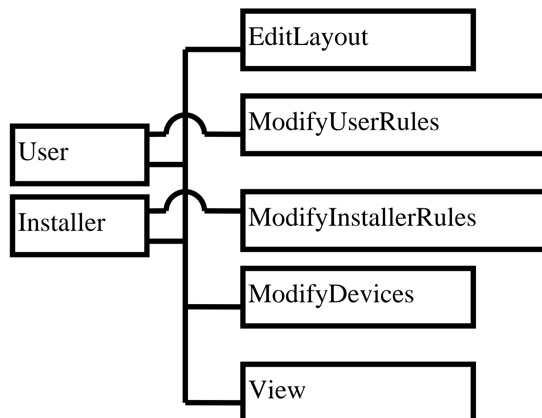


Customizing Capabilities

Both "User" and "Installer" are defaulted with the capabilities to perform all functions, and after installation, the User/Installer is free to either modify these or create new ones.

There are many ways to customize what is provided to fit your needs, but they fall into three general categories. First, you could keep the capabilities as provided and simply modify their assignments to resources and then add your password. Second, you could simply delete some included capabilities (say, ModifyRules, e.g.) from User and add a password to User. Third, you could create your own Capabilities, customize them the way you want, and replace the ones included in User, again applying your password to each. This latter approach is described below.

Note again, that any given resource like a device or rule set, can only have a single named capability assigned to each of its resource capabilities. So, for example, suppose the installer wants to protect a set of rule sets, and yet permit the user to create/modify other rule sets. In this case, it would be best to delete "ModifyRules" (which permits access to ALL rule sets), and then create two new capabilities named (say) "ModifyInstallerRules", and "ModifyUserRules". Then, the capability to modify each individual rule set can be selectively to either of these named capabilities. Further, the Install/User capabilities might look like:



A similar approach can be used for devices and Layouts.

Examples:

The Johnsons are avid hunters and keep their firearms in a securely locked gun room. To ensure that no one breaks into the gun room unnoticed, they wired the room with door sensors and motion detectors and then created Savoy WebEngine devices and rule sets to monitor the room. By creating a capability called GunRoom, and assigning it to all aspects of the rule set as well as Modify/Delete capabilities of the devices,

they are assured that no one can disarm their monitoring system without the GunRoom capability to which they assigned a password that only they know.

When the Jones go on vacation, they permit their neighbor to monitor the interior of their home (both are Savoy WebEngine users). They would like to grant their neighbor the right to view the home, but not the right to change it in any way. So, Jones creates a Layout with view-only capabilities which excludes the capability of changing the layout properties. Hence, the Layout is sealed against modification and is given to the neighbors on a floppy disk for their use -- a friendly and secure way of saying you can look but don't touch.

As a third example, an installer of home networks would like to permit a potential customer to access the installer's home for a limited period of time to gain some experience with the system, but make sure that after the trial period is over, the permission is revoked. To do this, the installer creates a special capability and seals it into a Layout file that he emails to his customer. He then grants his server this special capability for the duration of the trial period, after which he simply removes the capability, thereby disabling the Layout file forever.

Capabilities Applied to Device Name-Scoping in Peer Networks

Name-scoping allows a Savoy WebEngine to control complex environments comprised of a practically unlimited number of servers interconnected by the Internet. Name-scoping confines the network activity to localized regions, permitting more global sharing only when needed. With proper design, configurations of (say) hundreds or thousands of servers could be interconnected and globally controlled.

The server component of the WebEngine can be connected to another server, and can have many servers connected to it, forming a potentially large hierarchical network of peer servers. Assertions generated at any server can be propagated up and down all branches of the tree.

Assertions have the form `DeviceName AssertedValue`, and cooperation among servers is accomplished by shared device names. Device Name-scoping is a method of defining the space in which a device name is shared so that assertions for any given device does not propagate outside of the 'scope' of the device. This has the benefit of allowing the Savoy WebEngine to (1) manage very large networks of servers without limitations imposed by broadcasting all device assertions, and (2) allow the inter-operation of facilities each having independent device names.

Name-scoping in the WebEngine parallels Name-scoping in block structured computer languages such as C and Pascal. In these, a name is scoped by a block (defined by '`{…}`' in C, and `Begin…End` in Pascal) in which the name is defined. Outside of the block, the name is said to be 'out-of-scope' and within the block, it is 'in-scope'.

Each device in the Savoy WebEngine can be assigned a named scope. For example, Lamp1 might be assigned scope 'Private' whereas device FrontWalk might be assigned scope 'Global'. You can create as many scopes as you wish, and you may assign any number of devices to them.

In the WebEngine, scopes are represented by a named **Capability** (see previous section). The capability dialog (File Menu / Access) permits the construction of a set of capabilities possessed by the server.

When two servers connect, they exchange a list of devices. For each remote device presented to a server, if the server possess the capability of the device, a proxy for that device is created locally.

This can be made clear by the following example: Suppose we have multiple housing units per building and multiple buildings in a campus. We can create a hierarchy of servers following this organization and then install a set of capabilities for each server as follows:

- All servers have capability Campus.
- Each building server also has the capability unique to their building, like BldgOne.
- Each Unit server has a capability unique to all other units, like BldgOneUnitOne.

Devices created throughout the system are assigned a single capability that could be any one of the following: Campus, BldgOne, BldgTwo,…, BldgOneUnitOne, BldgOneUnitTwo, …, etc. As the severs connect, proxy devices are created if their capability is included in the server's list. For example, any device which is assigned Campus capability will be proxied on all machines.

Devices that are physically created on a local server, regardless of their scope, are termed *normal* devices. Devices that are within scope, but physically on another server are termed *proxy* devices. Devices that are out-of-scope and physically on another server are, of course, unknown to the local server. However, references to devices in rules that are unknown to the server are termed *virtual* devices.

Proxy devices differ from normal devices in that they do not communicate with any local Type Manager and no persistent context is maintained for them.

Archival Storage

All significant events that the system detects or actuates are stored in an archive file for possible future analysis. These events are accumulated over time in a file format that is structured to allow rapid access in reverse time order for one or more named events. We refer to this format as a *time-ordered-event*, or *toe* file.

Time Ordered Event (toe) format

A time-ordered-event (toe) file contains a sequence of time stamped events for a potentially large collection of event names. An example of an event name might be `GarageDoorOpen` and a toe file would include all events for this event name from the beginning of system operation to the present. Each such event name constitutes a *channel* in the file structure, and the archival system maintains independent chains of linked pointers for each such channel. The pointers provide rapid access through the file from the end (most recent) toward the beginning (past time) for each channel.

The principle advantage of this structure is to avoid sequential searches through a potentially very large file, particularly for events that are sparse in time (occur infrequently). During an orderly shutdown of the WebEngine, current pointer values are appended to the file which are subsequently reloaded during the next startup to provide continuity in the pointer chains. Occasionally, however, systems will fail and the pointer chains will be lost. When this occurs, the system will automatically recover the pointer chains through a salvaging procedure.

Reconstructing corrupted files

Reconstructing a damaged toe file can occur automatically during startup or explicitly by the user through a WebEngine menu command. In either case, normal operation of the system is temporarily suspended while it sequentially reads the entire toe file and reconstructs the lost pointer chains. Normal operation continues when this procedure is finished which typically takes a few minutes to perform.

Saving Event Data

Under the CyberHs application, a user can save a selected portion of event data for future off-line analysis, and these data files are also stored in toe format. The user is cautioned not to confuse file names, and overwrite the archival toe file. The CyberHs application checks this just in case.

Rule Builder

The Rule Builder is a convenient editor of rules. While rules are maintained by the WebEngine, the Rule Builder constructs and modifies them under the CyberHs viewer application, accessing the rules on the connected WebEngine.

The Rule Builder presents rules in the form of an English statement and allows the user to add/delete/modify rule sets, individual rules, and phrases within rules. The Rule Builder contains

dialog controls which automatically select any of the known devices, any of the known internal functions, and any of the known keywords, and so with occasional exceptions, the user can create or modify rules, in English, without prior knowledge of computer languages, and without typing, guaranteeing proper spelling and grammar.

Two rule builders are available; one, which requires minimal screen space, is designed to work graphically with device icons, and a second, which occupies a larger screen. Both are functionally equivalent, although the Graphical Rule Builder has more capabilities, like the ability to reorder the sequence of rules and assertions.

Graphical Rule Builder

The graphical rule builder can create rule components by right-mouse clicking on device icons. In its simplest form, it looks like:

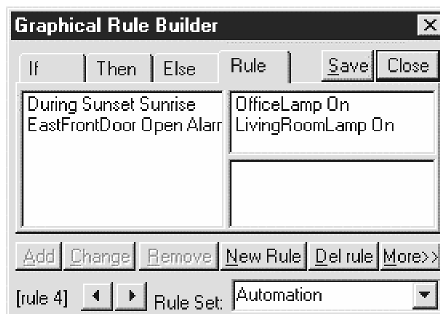


Figure 16: The Graphical Rule Builder (short form)

Clicking the More button, expands it to look like

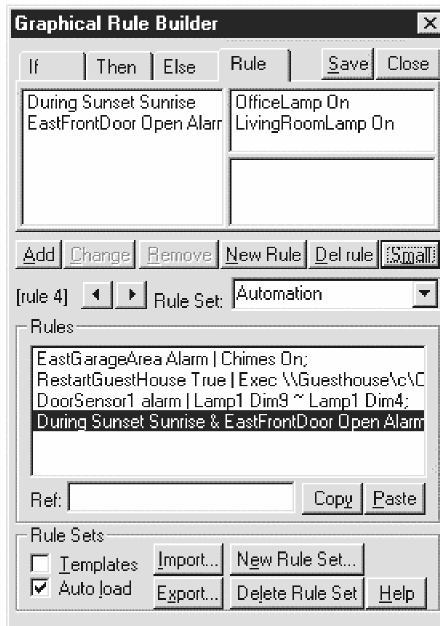


Figure 17: The Graphical Rule Builder (expanded form)

Basic Rule Builder

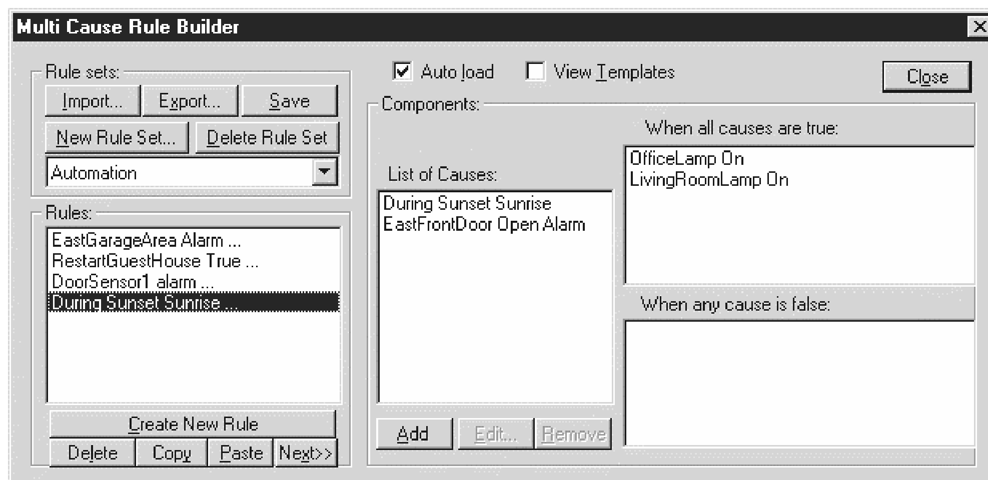


Figure 18: Rule Builder showing English sentence rules

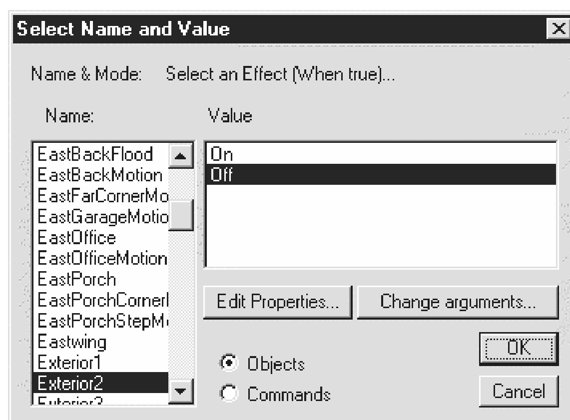


Figure 19: Either RuleBuilder showing component modification

WebEngine Applications

The term application here is misleading, because we do **not** mean a computer application. Rather, a *WebEngine Application* is defined as a set of devices and rule sets.

Applications permit users to:

- Back-up current devices and rule sets into named groups
- Share device/rule set groups among other users/machines
- Selectively implement & test subsystems, each an application

Users can create, load and save WebEngine Applications by selecting the the Savoy WebEngine File menu and clicking Server Applications… which presents the following dialog box.

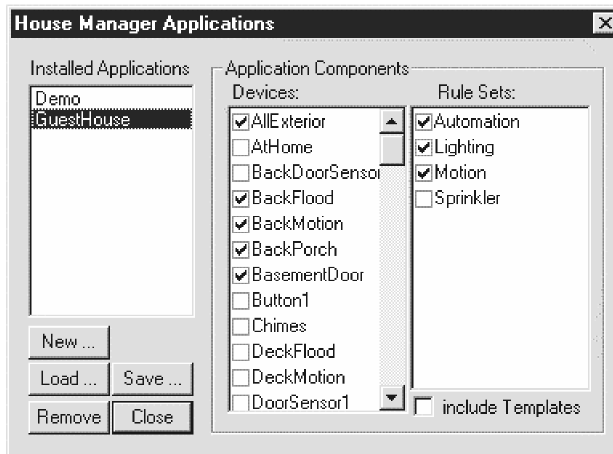


Figure 20: Server Application dialog box

Application components (devices and rule sets) are assigned to application names. These assignments are displayed when selecting an application (assigned components are checked). To change an assignment, simply alter the check mark and click Save to save the application to a file.

Once saved, applications can be transferred among the Savoy WebEngine systems and/or loaded back by clicking the Load button. Similarly, they can be totally removed by clicking the Remove button.

These operations cause considerable activity between the Savoy WebEngine client and the WebEngine server. For example, loading an application will cause new rule sets to be dynamically compiled, and if new devices are installed that were present in existing rule sets, they will be recompiled also. Removing an application will cause rule sets to be unlinked and destroyed. Virtual devices dynamically replace removed devices that are referenced in other rule sets. Reloading the application will cause these virtual devices to be replaced by the actual ones.

WebEngine Partitions

A single server computer can run multiple WebEngines each operating independently of each other. In such configurations, each WebEngine is said to run in a **partition**.

Partitions can be set up to be totally independent and protected from each other. Alternatively, partitions can be connected as peer servers as if they were distributed on a network, sharing device names and passing assertions among each other.

Each partition can operate an independent set of Type Managers. Partitioning permits multiple interfaces of the same type, such as multiple security panels, or multiple lighting systems. Examples of applications for partitioning include

- single installations that want multiple devices of the same type
- single servers that control multiple, independent facilities, such as an apartment building with multiple apartments.

Partitions are identified by a number. Single partition installations are defaulted as partition zero. You must ensure that each partition is set with a unique partition number; the WebEngine will not start if it detects another WebEngine running with the same partition number.

Each partition is a separate the Savoy WebEngine installation in a separate directory.

Installing Multiple Partitions

Partitioning requires TCP/IP protocol stack to be installed using the Network application in the Windows Control Panel.

Install he Savoy WebEngine into a separate directory for each position, beginning with the largest partition number and ending with partition zero. (This way, the shortcuts on the Start menu will be set to launch partition zero.)

Create shortcuts on the Start Menu, or on the desktop, to run the WebEngine in each partition – be sure to set the startup directory to the Savoy WebEngine directory for the partition.

Run the WebEngine for each partition and set the partition number from the Program Setup dialog.

Note that Layout Properties now include a partition number along with the TCP/IP address.

Sharing Across Partitions

You can configure multiple partitions to be independent, or to behave as a single entity. To do the latter, connect the partitions using peer-peer networking.

This may seem confusing – the generality provided by the networking architecture permits servers operating across a network (LAN or Internet) to behave as a single entity; the same mechanism can be applied to servers operating within the same computer, but in multiple partitions. When sharing across partitions, no actual network is involved, but everything operates as if they were distributed across a network. The reader is referred to earlier discussion on Network configurations starting on page 11.

Note that, similar to Layouts, WebEngine connections also include a partition number. So, to connect partition 1 to partition 0 on the same computer, use the local machine's IP address and simply specify the unique partition number.

As soon as partitions are connected, they immediately exchange device context, each creating **proxy** devices for devices that exist in other partitions. This permits rules to be written in any partition for all devices without regard to their (the device's) partition.

Network Applications of WebEngines

Savoy's Server module is a Win98/NT/CE application designed for monitoring and control of complex devices. Inherent within the module is the ability to combine them with other modules in a potentially large network. This section describes many of the networking features of the Savoy module and how they can be applied to solving real problems.

Devices and their Proxies

The principle entity with the Savoy system is the 'device', so named because much of what the Server does relates to actual real-world devices. However, the concept of a 'device' is considerably broader and includes many of the data types found in other programming environments. These are termed 'logical devices' in Savoy and include Boolean, Strings, and a variety of others. Devices, then, represent the fundamental 'name space' of the system and the capabilities of the devices represent the total capability of the server.

When two or more servers interconnect, they exchange device information and create 'proxy' devices that represent devices located on a connected server. This has two implications:

All devices across the network are in the same address space

When connected, each server has access to all devices, regardless of where they are.

Assertions to Devices Propagate across the Network

Assertions are changes of device state that originate from devices, rule engine(s), Java scripts, client applications, and so on.

Since assertions are automatically routed across the network, rules can be written to run in any server and control devices on any other server.

When authoring rules, it is recommended that the 'cause' side of the rules be on the server that hosts the referenced device, whereas the 'effect' side of the rule can apply to any device, local or not. This restriction does not apply to logical devices.

Scalability and Name Scoping

The concept of creating proxies for all remote devices has the obvious limitation of growing to very large numbers -- roughly as the square of the number of servers.

To prevent this limitation and permit the number of interconnected server to grow arbitrarily large, Savoy has implemented a concept of 'name scoping'. Name scoping is similar to that found in programming languages and is designed to contain the name of a device within a

certain 'scope'. The WebEngine implements a very general mechanism that functions as follows:

1. Each device is assigned a Scope Name -- a simple string, like 'Campus', or 'Building3'.
2. Each Server has a list of Scope Names.
3. Proxies are created on Servers ONLY IF that server posesses the device's Scope Name.

Imagine a network of Servers arranged on a map, and then think of a Scope as being singly-connected area on the map. Singly-connected means that there should only be a single area of any given name, not two or more areas.

There can be many Scopes and they can overlap arbitrarily. You can arrange them hierarchically (one Scope being totally contained within another) or not (intersecting Scopes).

To permit large networks, devices having broad scope should be few in number. Devices having narrow scope can be large in number. In other words, within any given Server, assign scope names so that most devices have local scope, a small percentage has broader scope, and only a few has global scope, for example.

Connecting Servers

A network of Servers is configured by two simple techniques:

1. Each WebEngine can connect to a single parent.
2. Each WebEngine can be the parent of many children.

Servers can connect to parents by configuration using the Setup dialog, or it can be done dynamically via rules. Connections are always initiated by the child, and so each Server need only know the IP Address of its parent.

It should be emphasized that the terms 'parent' and 'child' refer to the topology of interconnect and conveys no meaning as to how the Servers interact. In fact, once connected, all Servers interact as peers.

Connections are persistent and reliable. For example, individual Servers can be shut down momentarily and the network is automatically re-established on power up.

Client Applications

Savoy provides a variety of client applications together with software components that permit easy development of customer applications. In general, client applications can connect to any Server and see all devices that are in-scope. Clients can also originate assertions to control devices and receive assertions to change the clients rendering of device state.

The CyberHs Development Application

Unlike the WebEngine, the CyberHs application need not run all the time. When not running, all automation functions are performed by the WebEngine. The CyberHs application is required only when the user wants to view the system.

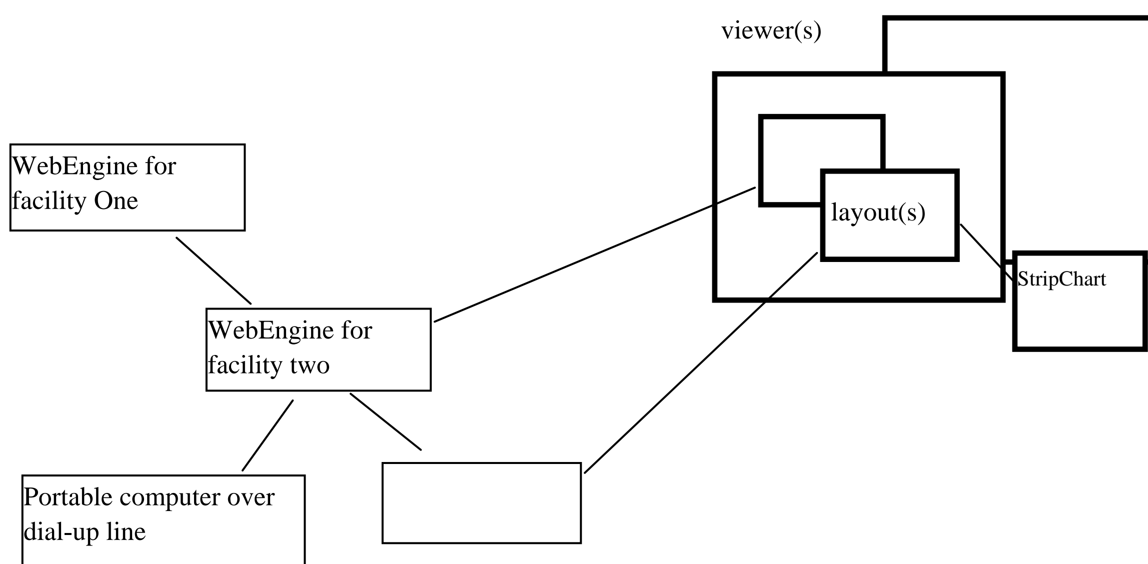


Figure 21: Savoy WebEngine component overview

Relationship to WebEngine

The CyberHs application is a client application of one or more WebEngines. Users can open one or more windows, called layouts, each of which can connect to the same, or different, WebEngine(s). Layouts typically provide a spatial view of a facility with controls to actuate and monitor device activity.

Each layout can have a related Strip Chart. These provide a temporal view of a facility giving the user a time chart of historical events, analogous to a paper strip chart recorder. The data used to render the charts can come from a WebEngine (live), or from a previously saved toe file. Portions of the Strip Chart can be selected and saved into a toe file for later analysis.

Layouts (Spatial view)

Layouts are child windows of the CyberHs application intended to provide a spatial view of the facility managed by the related WebEngine. A layout can connect to only a single WebEngine; however, there can be as many layouts within the CyberHs application as you want.

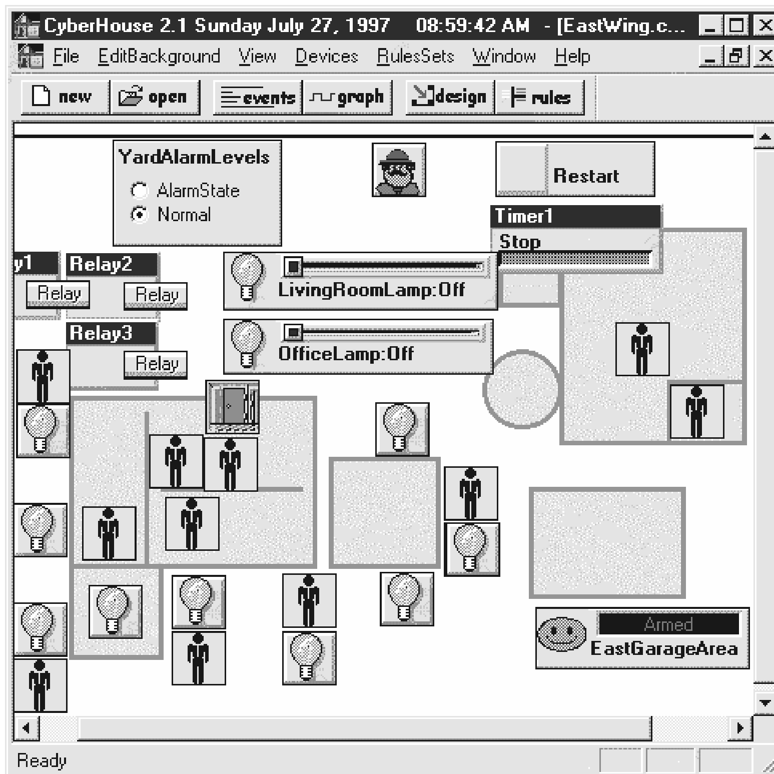


Figure 22: The CyberHs application's desktop with a layout displayed

Layouts are OLE2.0 clients. Consequently, the background of a layout can contain objects from any OLE2.0 server application including Microsoft Drawings, Charts, Excel spreadsheets, equations, audio sequences, video sequences, and many more. Many third party applications support OLE including architectural drawing packages such as AutoCad which provide three dimensional perspective views. Photographs input from scanners or digital cameras can be directly pasted onto the layout.

On top of this background are placed iconic representations of Savoy WebEngine controls. These controls allow the user to actuate devices and monitor activity. Their style includes a variety of representations including the extensive use of animations to provide accurate metaphors for the actual facility. For example, a motion detector control might animate a person walking when alarmed. A smoke detector might animate a fire burning, and so on. (see the *Device Context* section on page 69).

Local vs. networked

When setting up a layout for WebEngine connection, the user specifies whether the WebEngine is running on the same, or 'local', computer or whether it is accessed over a network.

In the case of networked access, the user is required to enter the name of the remote computer running the WebEngine. This name is the Computer Name as is specified in the Network utility within the Windows 95 Control Panel, or an Internet address such as 123.45.67.89, or savoysoft.com.

Strip Charts (Temporal view)

A single Strip Chart window can be created for each layout in the CyberHs application. These windows are similar in function to an actual paper strip chart recorder in that they display timed events, left to right, of one or more controls, each having its own 'pen'.

The Strip Chart can view events live, as they happen, or can access historical data for analysis. Live event monitoring is indicated by the LED on the 'pen bar', and enabled by clicking on it.

The Strip Chart defaults to displaying live event data when it is created, and this mode is retained unless the user *pans* the time line for later time spans. Later, the user can pan back to current time and see all the events that have since occurred while synching the Strip Chart to display subsequent live events (live sync is indicated by a red LED lamp on the pen bar).

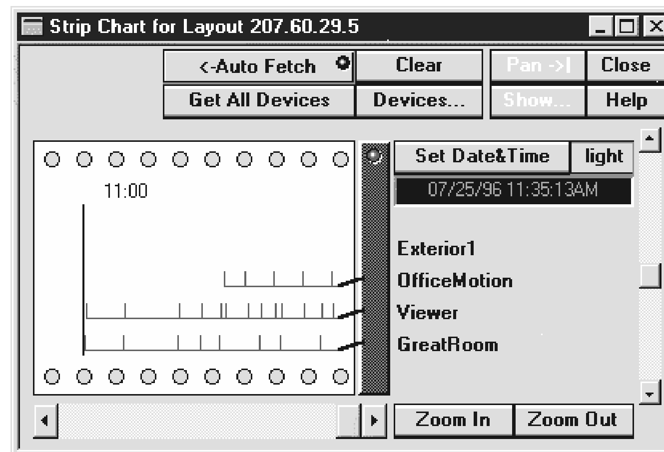


Figure 23: Strip Chart example

Historic data can be displayed for a selected time interval and for selected devices through the dialogs created by pushing the Devices button.

Data is retrieved from the WebEngine in time reversed order, and in blocks of about twenty events at a time. Each block received updates the display and automatically posts a request for the next block. This continues until the user releases the button labeled
<-Auto Fetch or a real event occurs which preempts the automatic fetch procedure. After the real event is received, the user can resume the fetch from where it left of by repressing the
<-Auto Fetch button again.

Once displayed, portions of the data can be selected by mouse movement and then saved into a toe file for future (off-line) analysis.

Device Context

The *context* of a Savoy WebEngine control includes its physical properties (type, address settings, modes, etc.) as well as the style properties used to display the control. The former

(physical properties) are exclusively managed by the WebEngine, whereas the latter (style properties) are exclusively managed by the CyberHs application. The name of the control, which must be unique for each control across the entire network, links the two contexts together.

Physical Properties

Physical properties can be access and modified from the CyberHs application, but they are stored and maintained by the WebEngine. The CyberHs application simply provides a convenient interface to the WebEngine, especially convenient when accessing the WebEngine remotely.

Physical properties are dependent on the type of device. For X-10 devices, it is usually only the House Code and Device Code that must be specified. For wireless devices, one need specify address information, supervision intervals, normally open/closed contact options, and so on. (Refer to the help system for more detail and other options).

Style Properties

Style properties are accessed through the CyberHs application, and also stored and managed by the CyberHs application. Style properties are associated with a specific layout and they follow the context of the layout during export and import. The same control can be displayed on two different layouts with entirely different styles, as designated by the user. Style properties are accessed by placing the mouse curson over a device's icon and clicking the secondary (right) button on the mouse to view the device's shortcut menu. Select Style from the menu. (Note that some device icons do not support style changes.)

Styles are configured by combining the choice of an *animator* and a *sequence*. These can then be applied to any of the standard *types* of display options.

What are *animators* and *sequences*? An animator solely specifies the timing characteristic of an animation, whereas a sequence is a set of images each representing a frame, as in a movie frame. Animators and sequences can be combined to provide a rich variety of options.

Animators

Standard animators provided include half cycle, full cycle, half/full cycle repeat, short cycle, and track value. They generically specify the time behavior of the control when its state changes (e.g., alarmed or on). For example, half cycle performs the animation in a forward only direction; full cycle goes forward and then reverses back to the beginning. A repeat half/full cycle continues the animation until the state reverts to normal (e.g., clear or off). A short cycle animation runs 'almost' full cycle, leaving the display 'dirty' with the next to last frame so the user can discern past activity. A track value animator goes up and down the animation in sync with a changing value of a control, like the dim level of an X-10 lamp module.

All of these animators can be independently applied to any of the following sequences.

Sequences

A sequence is an ordered set of icons, design so that when rapidly display, they show motion. Sequences typically have between two and sixteen (max.) individual frames. Many sequences

are provided with the system, and a user familiar with an icon editor can readily add their own. Consequently, users can customize their animations to best suit their application.

Types

Standard types of display renderings include various forms of colored boxes which animate by changing colors or size, as well as buttons, simple icons, icons overlaid on buttons, tabs which provide a slider bar control, displays which contain text received in a message, and so on. This standard set will be expanded to more forms as the need becomes apparent. Any of these types can be combine with any animator/sequence pair.

Savoy Console Application

The Savoy Console is an application that can be configured to run a combination of application plug-ins, which are ActiveX components designed for a specific function.

At a minimum, Plug-in functions include WebEngine connections and User authentication. Beyond these, examples include Rule Editors, Graphical Layouts, Domain Gateways (Patrol, Andover…), Console functions, and more.

The motivation behind Console is that User authentication and Server connection will be common to all applications and the Console provides these in a common framework. User authentication can be complex, with multiple levels of user capability and be subject to OEM customization itself -- possible by simply replacing the 'Authenticate' Plug-in. For Server connection, the Console permits concurrent connections to multiple clients.

Console plug-in applications are easy to create using Microsoft Visual C++ for ActiveX controls. Once installed on a user's computer, they can be combined with other plug-ins and run under the Console. Plug-in applications communicate with each other using a simple API supported by the Console environment.

Operation

After starting the Console, all users must log in (if enabled) by responding to the following dialog:

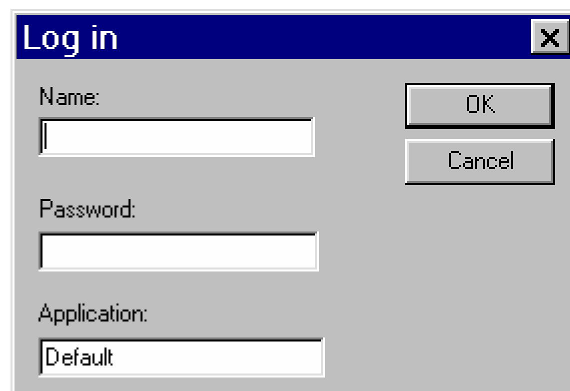


Figure 24 SPC login

To configure SCP, click the Setup button:

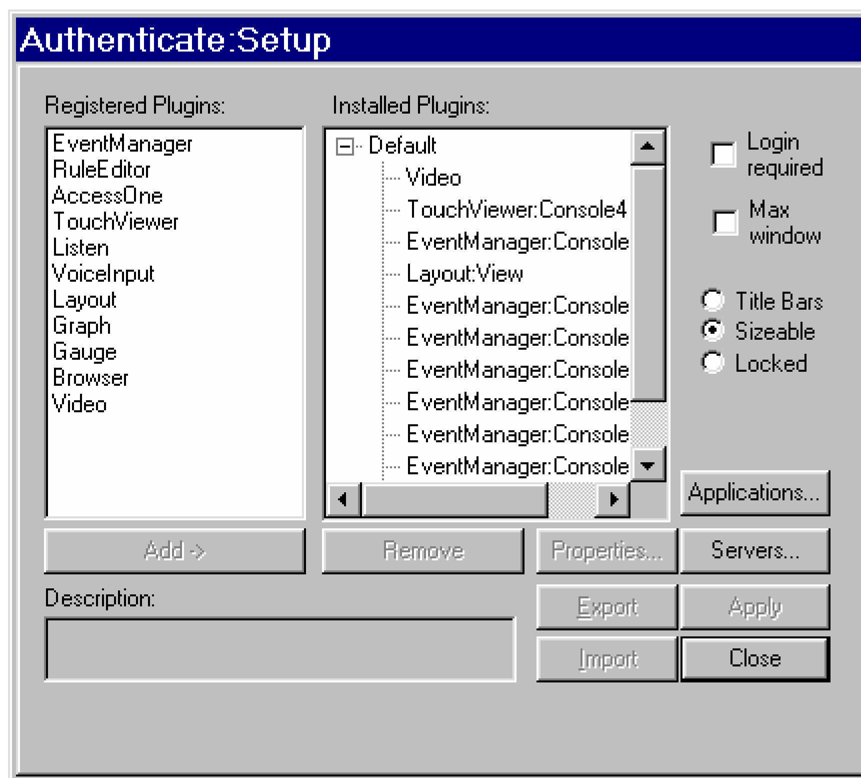


Figure 25 SPC Setup dialog

In the left box is a list of registered SPC plug-ins which can be installed by simply selecting them and clicking the Add button which will cause them to be listed in the Installed box. Installed plug-ins can be removed by selecting them and clicking the Remove button.

Description of Specific Plug-ins

Event Manager

When an Event Manager is added to the list of plug-ins, the framework prompts the user for a Event Manager instance name. This permits several Event Managers to run concurrently.

To control an Event Manager, you must create a logical text device having the same name as the Event Manager's Instance name. Events are sent to the Event Manager dialog by asserting values to the logical text device.

Current format of assertion value to the device is a series of flags followed by a text string. Example is:

/d an event /p 2 /c foo2 /r rules

This specifies a description (/d), a priority (/p), a context name (/c) and a ruleset template name (/r).

When the Event Manager plug-in receives this assertion, it adds the components to a list shown on the display, and then creates a temporary logical device named 'foo2'. This device will in turn create a ruleset from the template names 'rules', which will direct the procedural logic of the event.

An example of an Event template is:

^1 next:'Next'done:'Done''this is start';

^1 next | ^1 done:'Done''this is next';

^1 done | ^1 ok to remove...;

Note that the first line is a Declarative, executed when the ruleset is invoked. This places the event in an initial state. The remaining lines are rules that have specific state names as conditions and a text string that is a corresponding assertion. The format of the assertion is a series of Label-State pairs, where Label is typically a button label, and State is the state asserted when the button is pushed. These pairs are repeated on a single line, as in

DeviceName [NextState:'ButtonLabel'] [NextState:'ButtonLabel'] 'Message text...'

At the end of the line, only the text is given and this is displayed in a large text box.

Examples:

To ask a question,

DeviceName YesState:'Yes' 'Would you like to...?'

For a multiple choice,

DeviceName Button A:'A...' Button B:'B...' Button C:'C...' 'Please select from the following:'

Notes

If you have comments or questions, contact us at:

Savoy WebEngines, Inc.

Suite 3 ⬩ 30 Lyman Street

Westborough, MA 01581

800 ⬩ 527 ⬩ 2853

http://www.savoysoft.com