Chapter 11: Component model for Software Plug and Play

This chapter discusses the contributions from this thesis. It introduces and elaborates the component model for software plug and play that has been arrived at based on the empirical study of the experiments discussed in chapters 5 to 10. While, it is possible to identify a number of architectural elements and design patterns from these experiments, the focus of this thesis is on identifying those architectural abstractions that contributes to Plug and Play based extensible architectures. The objective is to use these architectural abstractions to formulate a Software Plug and Play component model. The component framework that is defined by this component model would facilitate interoperability, extensibility, customization, dynamic composition, service invocation, eventing, data exchange and deployment.

This chapter is organized as follows: a) The section on literature review will discuss about how plug and play is presently supported, b) The section on hypothesis and observations will present the lessons learnt from the various experiments, c) The section on principles will discuss about the principles and guidelines that are culled out from the various experiments, d) The section on abstractions will discuss about the plug and play architectural abstractions that are culled out from the various experiments, e) The section on the component model will discuss about how all these have been put together, f) The last section will illustrate the use of the component model.

11.1 Review of Literature on how Plug and Play is enabled

There are different approaches adopted in the industry to support Plug and Play. Most of these approaches have been found to be practically useful and some are arrived at based on an industry level agreement. Some approaches select from a library of configurations of components and connectors and use the one that best fits the current availability of components and connectors. Some other approaches maintain a catalogue of pre-defined connectors (drivers) and choose those that are best suited for establishing connectivity and control of the new component that is plugged in. Some approaches load a pre-defined schema file, which specifies the visibility of the components and the supported operations, during system start-up. It is evident that there is no common consensus about these approaches, but each of these designs has been practically useful. In the subsequent sections, a literature survey of the existing plug and play approaches is discussed.

According to the Universal Plug and Play (UPnP) forum [4-5], UPnP is enabled by the UPnP device architecture that defines the protocols for communication between control points (controllers) and devices. UPnP performs device and service discovery and control through driverless, standards based
protocols mechanisms. UPnP devices can automatically configure network addressing, announce their presence on a network and permit exchange of device and service descriptions. The various protocols in this device architecture are given below:

a) Addressing is Step 0 of UPnP. Through addressing, devices and control points get a network address. The foundation for UPnP networking is IP addressing.

b) Discovery is Step 1 of UPnP. Through discovery, control points find interesting devices. When a device is added to the network, the UPnP discovery protocol allows that device to advertise its services to control points on the network which is then found by the control points during the devices search.

c) Description is Step 2 of UPnP. Through description, the device and its capabilities are shared with the control point so that the control point can interact with the device. While the device description details out the physical and logical aspects, the service description describes the capabilities exposed by the device.

d) Control is Step 3 of UPnP. Based on the knowledge gained about a device and its capabilities (through description), a control point can ask those services to invoke actions and receive responses indicating the result of the action.

e) Eventing is Step 4 of UPnP. Eventing is linked with Control (Step 3) in such a way that when control points send actions to devices, through eventing, control points listen to state changes in devices. The two forms of eventing that are supported in UPnP are Unicast eventing and Multicast eventing.

f) Presentation is Step 5 of UPnP. Presentation exposes an HTML based user interface for controlling and/or viewing device status. The degree to which each of these can be accomplished depends on the specific capabilities of the presentation page and the device under consideration. Retrieving a presentation is a simple HTTP based process.

In essence, to enable Universal Plug and Play [4-5], handshaking at multiple levels between the device and the computer System so as to establish the identity of the device and standardized messaging and event handling mechanisms are deemed necessary.

According to Microsoft [9-11], Plug and Play requires support from device hardware, system software and operating system drivers that connect the device to the system. The device hardware support involves support for discovering the device, describing its capabilities, utilizing control points to leverage the device capabilities, responding to events that are generated due to certain actions triggered by the
control points and interfacing with the device to manipulate its configuration. The system software support for Plug and Play provides automatic and dynamic recognition of installed hardware based on device events like mount, unmounts, dock and undock; a Plug and Play manager to facilitate hardware resource allocation and reallocation; a Plug and Play device manager to determine and load the appropriate drivers; a programming interface for drivers to interact with the Plug and Play system; and Plug and Play mechanisms for drivers and applications to act on hardware events triggered by the actions on the device. The operating system drivers link physical or virtual devices with the operating system, making the devices available to the operating system and the applications through an interface. These drivers can be either stream interface drivers (implement and expose a standard set of stream interface functions) or native device drivers (implement and expose interfaces other than stream interfaces). In essence, to enable Plug and Play, Microsoft provides a comprehensive Windows driver development tool kit that supports functionalities needed at the device level, operating system level and the interface level.

According to Luckham et al [48-49], Plug and Play architectures are enabled by a specific configuration of interface connection architectures in which interfaces are structured by services that provide a connection between interface features. While interface connection architecture defines all connections between the modules of a software system using only the interfaces, each of these interfaces specifies both provided and required features and connections are defined between a required feature and a provided feature such that any use of a required feature is replaced by the use of the provided feature and calls to required feature in turn become calls to provided feature. In case of Plug and Play architectures, interfaces are allowed to provide services, and dual services may be connected together by a service connection (single connection that denotes a set of interface connections, one between each pair of dual features in the two services). A correct connection between the host components and the component that is plugged should satisfy the formal constraints on interface features and also the protocols relating dataflow between the features. In essence, Plug and Play architectures are dual interface connection architectures between the host configuration and the component that is plugged in.

Bronsard et al [126], lay down certain attributes that should be supported by software systems in order to enable Plug and Play. These Plug and Play attributes are:

a) Context Independence: Software components should state the assumptions made about their context and they should not know which other components they interact with.

b) Location Transparency: Software components should be part of multiple assemblies and they should also be in a position to be assembled based on different configurations.
c) Evolvability: It should be easy to add, remove, replace and modify components of an assembly.

d) Nature of components: Components can be either primitive or an assembly of other components.

e) Behaviour: The functionality of a component should be defined by its interfaces (provided interfaces specify those capabilities that a component offer, required interfaces specify those capabilities that a component receives from its environment).

f) Assembly/Configuration: Should contain the information needed to integrate heterogeneous components into an executable assembly.

11.2 Hypothesis, Observations and Insights

The hypothesis for the six graded experiments of varying complexity and the observations on their corresponding plug and play architectures are presented in Table 11.1. This table illustrates the usefulness of plug and play as an architectural abstraction for architecting extensible systems.

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 If components are dynamically attached and detached at run-time, there is no discrepancy in the system performance</td>
<td>Plug and Play architectures did not impact interface performance</td>
</tr>
<tr>
<td>2 If components that are dynamically attached and detached can maintain consistency relationships on the data they manipulate, then an integrated view of data can be created</td>
<td>Plug and Play architectures did not impact data integrity</td>
</tr>
<tr>
<td>3 If components are plug and playable, then irrespective of the domain, extensible systems become possible</td>
<td>Plug and Play architectures are domain independent</td>
</tr>
<tr>
<td>4 If components are plug and playable, then two different types of components can be plugged in without functional overlap</td>
<td>Plug and Play architectures are polymorphic</td>
</tr>
<tr>
<td>5 If a standardized plug and play mechanism is utilized and adopted to plug components of varying complexity, then consistent high performance is possible</td>
<td>Plug and Play architectures are high performant, and scalable</td>
</tr>
<tr>
<td>6 If the plug and play mechanism is simplified, then high performant scalable systems are possible</td>
<td>Plug and Play architectures are high performant, scalable and simple</td>
</tr>
</tbody>
</table>

Table 11.1: Hypothesis and Observations of the Experimental Systems

The observations listed in Table 11.1, lead to the following insights:

- Plug and play architectures are governed by a set of common principles.
Plug and play architectures utilizes an underlying component model to facilitate dynamic composition.

Plug and socket interfaces serves as the extensibility mechanisms for the component models.

Plug and socket interfaces are encoded in the run-time of the components/component models.

Plug and socket interfaces are implementation, domain and technology agnostic.

In the subsequent sections, these insights are detailed out to express the plug and play architectural abstractions.

### 11.3 Plug and Play Architecture Principles

Architecture principles define the underlying general rules and guidelines for the use of a component model [51]. They reflect the level of consensus among the various architecture elements and form the basis for decision making. In each of the experiments that were discussed earlier, a set of principles were found to remain true in enabling plug and play characteristics to the resultant architecture. Some of these principles that were adopted by the different experiments are taken up for characterizing Plug and Play architectures as presented in Table 11.2.

<table>
<thead>
<tr>
<th>Principle</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Principle of Simplicity</td>
<td>To reduce complexity, it is necessary to simplify the external components by reducing them to the bare essentials</td>
</tr>
<tr>
<td>Principle of Compatible support Elements</td>
<td>To reduce interface mismatch, it is necessary to allow only those elements that fit the interface specifications</td>
</tr>
<tr>
<td>Principle of Minimum Communication</td>
<td>To maintain information integrity, it is necessary to share information with only those components that are one level above or one level below in the component hierarchy</td>
</tr>
<tr>
<td>Principle of independent Partitioning</td>
<td>To reduce functionality overlap, it is necessary to choose external components that are as independent as possible</td>
</tr>
<tr>
<td>Principle of Replacement</td>
<td>To enable backward compatibility, it is necessary to choose external components that support new functionality over and above previous functionality</td>
</tr>
<tr>
<td>Principle of Low Complexity</td>
<td>To enable interfacing, it is necessary to obfuscate internal complexity and make the external interfaces simple</td>
</tr>
<tr>
<td>Principle of Least Privilege</td>
<td>To improve stability, external components should be designed to operate with least possible privileges.</td>
</tr>
<tr>
<td>Principle of Economy</td>
<td>To improve scalability, external components should be simple providing necessary functionality and nothing more or nothing less.</td>
</tr>
<tr>
<td>Principle of Localization</td>
<td>For better performance, external components should perform all local activities at high speed and all external activities at host component speeds.</td>
</tr>
<tr>
<td>Principle of Grouping</td>
<td>To facilitate functional extensibility, it is necessary for components that provide similar functionality should be grouped together.</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Principle of Similarity</td>
<td>To facilitate ease of usage, external components should adopt similar processes wherever possible.</td>
</tr>
<tr>
<td>Principle of Consistency</td>
<td>To maintain information integrity, consistency relationships should be maintained between the external components</td>
</tr>
<tr>
<td>Principle of Understanding</td>
<td>To maintain ease of understanding, it is necessary to obfuscate unnecessary details of the external component and focus on the essentials</td>
</tr>
</tbody>
</table>

**TABLE 11.2: SOME USEFUL ARCHITECTURE PRINCIPLES FOR PLUG AND PLAY**

### 11.4 Plug and Play Architecture Abstractions

Architecture abstractions define common elements and their configurations that arises to address the desired function. It is through these abstractions that architectures can be characterised. From the experiments discussed earlier, a set of architectural abstractions that were found to be useful were identified. In the subsequent sections, these abstractions are used to characterize the Plug and Play component model. They are considered, based on the premise that “Software Plug and Play is a collection of standards, encoded in the run-time and embedded in the component schema, in order to support functional and structural extensibility of the instrumentation platform”.

#### 11.4.1 Plug

A plug comprises of a collection of capabilities that enables the plugged component to be executed on an underlying computing system. It serves as the gateway to the plugged component, for accessing the computing resources provided by the component framework. The plug encodes the interface mechanism for connecting to a compatible socket thereby enabling the plugged component to function as an external component of the component framework. The plug is implemented as part of the run-time support of the plugged component.

#### 11.4.2 Socket

A socket is a handle that allows the plugged components to utilize the computing resources made available by the component framework. It serves as the gateway to the component framework, for accessing the computing resources provided by the plugged components. Each socket can support zero or more plugged components. The socket encodes the interface mechanism for connecting to a plug and performs type checking to validate the compatibility of the plug. The socket is implemented as part of the run-time support of the component framework.
11.4.3 Component Framework

A component framework defines specific interaction and composition standards that are adopted by the constituent components and provides the set of elements required for supporting the execution of these components. It implements a component schema which describes the characteristics of the constituent components that it should support. It provides computing resources which are accessible to the plugged components through the socket and it consumes the computing resources provided by the plugged components through their plugs.

11.4.4 External Component

An external component extends and adds functionality to the component framework thereby enabling the evolution of the underlying system. It utilizes the services provided by the Plug to attach itself, as a plugged component, to the component framework. It utilizes the computing resources provided by the component framework and provides computing resources for utilization by the component framework. A plugged component is designed to operate locally within its state space. For any additional requirements, it utilizes the resources provided by the component framework.

11.4.5 Event

An event is an encapsulated set of actions that are triggered during the utilization of the computing resources when certain pre-defined conditions are met. Some of the events that are triggered inside the plugged components are handled locally and the rest dispatched to the component framework for resolution. Similarly, some of the events that are triggered inside the component framework that are relevant to the plugged components are dispatched to them and the rest dispatched to its constituent components. It is often the case that the event information is supplemented with additional event data thereby enabling better resolution of the event.

11.4.6 Service

A service is an encapsulated set of functionalities offered by a component, along with a set of policies of usage that declare the information necessary for the successful execution of the service, that can be invoked by other components for different purposes. Services needs to be first discovered and then invoked using appropriate mechanism. It is often the case that some services offered by the plugged components are consumed by the component framework and some services offered by the component framework are consumed by the plugged components. It is often the case that information needs to be
provided for the invocation of services and this is either made possible by the use of an additional data channel or by the use of service parameters.

11.4.7 Task

A task corresponds to a set of actions that are performed by a component in order to achieve a purpose. The task is the minimal unit of work that can be assigned to a component. While performing the task, additional meta-data information is made available so as to complete the work. The success or failure of performing the task is often utilized to make decisions on future course of action. Tasks are often hierarchical in nature, in the sense that component frameworks assign tasks to plugged components, and monitor their completion.

11.4.8 Configuration

A configuration is an encapsulated set of preferences supported by a component on a set of computing resources that it provides. By changing the configuration of the component, it becomes possible to change its behaviour thereby enabling customization of a component. A component’s preferences are often dictated by a higher purpose and in the case of the Plug and Play architecture, the component framework determines the desired configuration of the plugged components.

11.4.9 Information

In computing, information is a fundamental entity and is often considered as data that is processed, stored and transmitted by computing systems. Without exchange of information, the interaction between components becomes meaningless. The Plug and Socket serve as information pathways for exchanging different types of information between the plugged components and the component framework. As a result, these pathways serve as the backbone of the Plug and Play architecture.

11.4.10 Repository

A repository is an archive that has the responsibility to preserve information and make it available for a designated component. It is often considered as a receptacle in which things are stored. In the plug and play architecture, because of the fluid nature of the connectivity between components, it is preferable to maintain a repository which hosts the information that the different components accesses and manipulates. As a result, it is maintained at the same level as the component framework so that information is not lost when the external components are disconnected.
11.5 *Generic Plug and Play Component Model*

The generic Plug and play component model [127] is put together by considering the plug and play architectural abstractions as well as the architecture principles that were discussed in the earlier sections 11.4 and 11.5. Accordingly, the constituent elements of the generic Plug and play component model are:

a) Component framework which implements the component schema.
b) Component framework run-time which serves as the socket.
c) Repository which serves as a receptacle of information.
d) External components which provide new capabilities and are plugged in at run-time.
e) External components run-time which serves as the Plug.

A set of agreed upon standards that are necessary to support the different capabilities needed by the external components and the component framework.

![Diagram of Generic Plug and Play Component Model]

**Figure 11.1: Decomposition of Generic Plug and Play Component Model**

a) A collection of protocols that are necessary to support the communication between the different components.
b) A set of channels that are necessary to support the flow of information during the communication between components.

c) A set of events that are listened to and acted upon during the execution of the components. These events are predominantly technology specific.

d) A set of services that are provided and consumed by the different components during execution based on the desired functionality.

Figure 11.1 provides a decomposition of this model. Since the focus of this thesis is on Plug and Play, the model does not bring forth the other functional and non-functional characteristics of the component model. The focus is exclusively on how plug and play is enabled. Additionally, for the purposes of this discussion, only one external component and its associated run-time is depicted, for all practical purposes more than one external component and associated run-time can be present. The subsequent sections provide a brief overview of some of the implementation specific constructs.
11.5.1 Channels

There are three channels that are of importance with regard to communication between the component framework and the external components are: a) Event channel which provides the pathways necessary for dispatching events to the component framework and the external components, b) Configuration channel which provides the pathways necessary for modifying the external component configuration and, c) Data channel which provides the pathways necessary for exchanging information between the component framework and the external components.

11.5.2 Standards and Protocols

The generic Plug and play component model utilizes a variety of protocols and standards for establishing connectivity and communication. Standards provide an agreed upon way of supporting capabilities in components and enable components to interoperate, exchange information and support a
wide variety of interfaces. Protocols provide rules and constraints governing the communication between
two or more components and assure information integrity, security and availability. Figure 11.2, provides
an illustration of the different protocols and standards that are useful in this situation.

11.5.3 Services and Events

The generic Plug and play component model utilizes a variety of services and events for facilitating
capability utilization. Figure 11.3 provides a simplistic view of the services and events that facilitates plug
and play. The various events and corresponding services could be summarized as:

a) Discovery is Step 0 of software Plug and Play. Through discovery, components are identified and
registered with the socket.

b) Identity is Step 1 of software Plug and Play. Through identity, sockets obtain interesting services
and functions that are provided by the plugs. When a Plug is added to the socket, the identification
protocol allows that plug to register itself and enumerate its services.

c) Control is Step 2 of software Plug and Play. Based on the knowledge gained about a plug and its
capabilities, the socket can invoke the various services/functions and receive responses indicating
the result of the action.

d) Eventing is Step 3 of software Plug and Play. Eventing is linked with Control (Step 2) in such a way
that when sockets request plugs to perform certain service, a state change is triggered.

e) Presentation is Step 4 of software Plug and Play. Presentation exposes the information generated
by the plug and socket interaction. The degree to which the information is presented depends on
the specific capabilities of the presentation page and the plug under consideration.

11.6 Abstract Representation of Plug and Socket

In this thesis, it could be observed that six graded experiments corresponding to difference class of
problems have been served by plug and play software architectures. Each experiment discussed in
chapters 5 to 10 express one class of problems. Each of these experiments is supported by a run-time that
corresponds to the problem space. This run-time characterizes a class of interface components with
facilities for access and manipulation. In order for the interface to be established successfully, it is
necessary that there is type matching between the plug and socket. To assert that this interface is
independent of its implementation, this thesis utilizes abstract data types to express the interfaces. In
the subsequent sections, the plug and socket abstract data types are discussed.
11.6.1 Plug Abstract Data Type

The Plug (identified by the Sort Plug) provides services, functionalities, and resources necessary for treating the external component as a composable component. It serves as a composition mechanism that provides execution and composition services to the external component. It utilizes the preferences (identified by the sort Configuration) to control the external component’s behaviour. It utilizes the data (identified by the sort Data) to exchange information between the external component and the component framework. It listens to and dispatches events (identified by the sort Event) triggered by the computing system to appropriate event handlers. It provides a plug interface which is a corollary for the socket in the component framework. Its abstract data type is given in Table 11.3.

<table>
<thead>
<tr>
<th>Sort</th>
<th>Plug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imports</td>
<td>Data, Configuration, Event, Service, Data Channel (DC), Event Channel (EC), Configuration Channel (CC), Service Collection (SC), Event Collection (EVC)</td>
</tr>
<tr>
<td>Operations</td>
<td>Initialize (DC, EC, CC) → Plug</td>
</tr>
<tr>
<td></td>
<td>Register (Plug, Socket) → Plug</td>
</tr>
<tr>
<td></td>
<td>Unregister (Plug) → Plug</td>
</tr>
<tr>
<td></td>
<td>IsRegistered (Plug) → Boolean</td>
</tr>
<tr>
<td></td>
<td>Specify (Plug) → Configuration</td>
</tr>
<tr>
<td></td>
<td>Set (Plug, Configuration) → Boolean</td>
</tr>
<tr>
<td></td>
<td>Get (Plug) → Data</td>
</tr>
<tr>
<td></td>
<td>Receive (Plug) → Data</td>
</tr>
<tr>
<td></td>
<td>Put (Plug, Data) → Boolean</td>
</tr>
<tr>
<td></td>
<td>Send (Plug, Data) → Boolean</td>
</tr>
<tr>
<td></td>
<td>Discover (Plug) → SC</td>
</tr>
<tr>
<td></td>
<td>Enumerate (Plug) → SC</td>
</tr>
<tr>
<td></td>
<td>Locate (SC, Integer) → Service</td>
</tr>
<tr>
<td></td>
<td>Count (SC) → Integer</td>
</tr>
<tr>
<td></td>
<td>Provide (Plug, Service) → Boolean</td>
</tr>
<tr>
<td></td>
<td>Consume (Plug, Service) → Boolean</td>
</tr>
<tr>
<td></td>
<td>Identify (Plug) → EVC</td>
</tr>
<tr>
<td></td>
<td>Detect (Plug) → EVC</td>
</tr>
<tr>
<td></td>
<td>Find (EVC, Integer) → Event</td>
</tr>
<tr>
<td></td>
<td>GetSize (EVC) → Integer</td>
</tr>
<tr>
<td></td>
<td>Listen (Plug, Event) → Boolean</td>
</tr>
<tr>
<td></td>
<td>Dispatch (Plug, Event) → Boolean</td>
</tr>
<tr>
<td></td>
<td>Handle (Plug, Event) → Boolean</td>
</tr>
<tr>
<td>Axioms</td>
<td>IsRegistered (Initialize (DC, EC, CC)) = False</td>
</tr>
<tr>
<td></td>
<td>IsRegistered (Register (Plug, Socket)) = True</td>
</tr>
<tr>
<td></td>
<td>IsRegistered (Unregister (Plug)) = False</td>
</tr>
</tbody>
</table>
11.6.2 Socket Abstract Data Type

The Socket (identified by the Sort Socket) provides services, functionalities, and resources necessary for composing the external components with the component framework. It can handle more than one plug that satisfies the interface requirements of the socket. It serves as an integration mechanism for all the external components. It sends configuration information (identified by the sort Configuration) to the plug which is then used to update the external component configuration. It utilizes the information exchange services provided by the Data (identified by the sort Data) to share data between the component framework and the external components. It listens to and dispatches events (identified by the sort Event) triggered by the computing system to appropriate handlers. The abstract data type of the socket is given in Table 11.4.
- Send (Socket, Integer, Data) → Boolean
- Receive (Socket, Integer) → Data
- Discover (Socket, Integer) → SC
- Detect (Socket) → SC
- GetEvent (SC, Integer) → Service
- GetCount (SC) → Integer
- Provide (Socket, Integer, Service) → Boolean
- Consume (Socket, Integer, Service) → Boolean
- Identify (Socket, Integer) → EVC
- Generate (Socket) → EVC
- GetEvent (SC, Integer) → Service
- GetCount (SC) → Integer
- Provide (Socket, Integer, Service) → Boolean
- Consume (Socket, Integer, Service) → Boolean
- Identify (Socket, Integer) → EVC
- Generate (Socket) → EVC
- Find (EVC, Integer) → Event
- GetSize (EVC) → Integer
- Listen (Socket, Integer, Event) → Boolean
- Dispatch (Socket, Integer, Event) → Boolean
- Handle (Socket, Event) → Boolean

**Axioms**
- Count (Initialize (DC, EC, CC)) = 0
- Count (Add (Socket, Plug)) = Count (Socket) + 1
- Count (Remove (Socket, Integer)) = Count (Socket) – 1, if Count (Socket) > 0
- IsRegistered (Socket, Integer) = False, if Integer > Count (Socket)
- Set (Socket, Integer, Configuration) = False, if IsRegistered (Socket, Integer) = False
- Send (Socket, Integer, Data) = False, if IsRegistered (Socket, Integer) = False
- Receive (Socket, Integer) = Undefined, if IsRegistered (Socket, Integer) = False
- Discover (Socket, Integer) = Undefined, if IsRegistered (Socket, Integer) = False
- GetEvent (SC, Integer) = Undefined, if Integer > GetCount (SC)
- Provide (Socket, Integer, Service) = False, if IsRegistered (Socket, Integer) = False
- Consume (Socket, Integer, Service) = False, if IsRegistered (Socket, Integer) = False
- Identify (Socket, Integer) = Undefined, if IsRegistered (Socket, Integer) = False
- Find (EVC, Integer) = Undefined, if Integer > GetSize (EVC)
- Listen (Socket, Integer, Event) = False, if IsRegistered (Socket, Integer) = False
- Dispatch (Socket, Integer, Event) = False, if IsRegistered (Socket, Integer) = False

**Table 11.4: Socket Abstract Data Type**

### 11.7 Simple illustration of Plug and Play

The purpose of this illustration is to show the difference that Plug and Play brings to the architecture. The component model comprises of four different components (C1 to C4) and its associated run-time. For this discussion, component diagrams are used to express the model so that it becomes easy to understand the overall structure. For the sake of simplicity, the different component interfaces are not included in the illustration, however, the relationships between the components are illustrated. Figure 11.4 depicts the typical component model without plug and play support.
In this setup, if the component C1 is treated as an external component then, the resultant plug and
play enabled architecture would be as illustrated in Figure 11.5.
Additionally, different variants of the external component C1 can be plugged to this structure as illustrated in Figure 11.6. The socket embedded in the run-time takes up the responsibility of managing the different plugs as well as handling the events, configuring the different external components, exchanging relevant data and providing/consuming appropriate services. This is the difference that the plug and play component model brings to an architecture.

11.8 Failed Plug and Play: Incompatible plug and socket

The purpose of this illustration is to depict a possible scenario where plug and play fails to occur due to incompatibility between the plug and the socket. The component model comprises of three different components (C1 to C3) and its associated run-time. C4 is considered to be the external component that is plug and playable. For this discussion, component diagrams are used to express the model so that it becomes easy to understand the overall structure. For the sake of simplicity, the different component interfaces are not included in the illustration, however, the relationships between the components are illustrated. Figure 11.7 depicts the plug and play enabled architecture.
The figure 11.7, illustrates the lacunae in expressing incompatible plug and socket which occurs while using component diagrams to express plug and play architectures. In a typical architecting endeavour, the effort that is expended is to assert that composition of all components is possible. In this case, where the plug and socket are incompatible, such a composition is not possible. However, the external component is plugged in after the system is implemented and during run-time. This incompatibility cannot be expressed using component diagrams as the nature of incompatibility is unknown to the architect. One way to address this problem is to use abstract data types to express algebras. The incompatibility between the plug and the socket, can be expressed as type incompatibility. The subsequent sections provide abstract data type specifications of incompatible plug and socket.

The Plug (identified by the Sort Plug) provides functionalities, and resources necessary for supporting plug and play. It utilizes preferences (identified by the sort Configuration) to control the external component’s behaviour. It utilizes the data (identified by the sort Data) to exchange information between the external component and the component framework. It invokes and allows invocation of the various functions (identified by the sort Function). It listens to and dispatches events (identified by the sort Event) triggered by the computing system to appropriate event handlers. Its abstract data type is given in Table 11.5.
Sort Plug

Imports Data, Configuration, Event, Function, Data Channel (DC), Event Channel (EC), Configuration Channel (CC), Function Collection (FC), Event Collection (EVC)

Operations
- Initialize (DC, EC, CC) → Plug
- Register (Plug, Socket) → Plug
- Unregister (Plug) → Plug
- IsRegistered (Plug) → Boolean
- Specify (Plug) → Configuration
- Set (Plug, Configuration) → Boolean
- Get (Plug) → Data
- Receive (Plug) → Data
- Put (Plug, Data) → Boolean
- Send (Plug, Data) → Boolean
- Discover (Plug) → FC
- Enumerate (Plug) → FC
- Locate (FC, Integer) → Function
- Count (FC) → Integer
- Invoke (Plug, Function) → Boolean
- RemotelInvoke (Plug, Function) → Boolean
- Identify (Plug) → EVC
- Detect (Plug) → EVC
- Find (EVC, Integer) → Event
- GetSize (EVC) → Integer
- Listen (Plug, Event) → Boolean
- Dispatch (Plug, Event) → Boolean
- Handle (Plug, Event) → Boolean

Axioms
- IsRegistered (Initialize (DC, EC, CC)) = False
- IsRegistered (Register (Plug, Socket)) = True
- IsRegistered (Unregister (Plug)) = False
- Specify (Plug) = Undefined, If IsRegistered (Plug) = False
- Set (Plug, Configuration) = False, If IsRegistered (Plug) = False
- Get (Plug) = Undefined, If IsRegistered (Plug) = False
- Receive (Plug) = Undefined, If IsRegistered (Plug) = False
- Put (Plug, Data) = False, If IsRegistered (Plug) = False
- Send (Plug, Data) = False, If IsRegistered (Plug) = False
- Discover (Plug) = Undefined, If IsRegistered (Plug) = False
- Enumerate (Plug) = Undefined, If IsRegistered (Plug) = False
- Locate (FC, Integer) = Undefined, If Count (FC) < Integer
- Invoke (Plug, Function) = False, If IsRegistered (Plug) = False
- RemotelInvoke (Plug, Function) = False, If IsRegistered (Plug) = False
- Identify (Plug) = Undefined, If IsRegistered (Plug) = False
- Detect (Plug) = Undefined, If IsRegistered (Plug) = False
The Socket (identified by the Sort `Socket`) provides services, and resources for supporting plug and play. It can handle more than one plug that satisfies the interface requirements of the socket. It serves as an integration mechanism for all the external components. It sends configuration information (identified by the sort `Configuration`) to the plug which is then used to update the external component configuration. It utilizes the information exchange services provided by the Data (identified by the sort `Data`) to share data between the component framework and the external components. It provides and consumes services (identified by the sort `Service`). It listens to and dispatches events (identified by the sort `Event`) triggered by the computing system to appropriate handlers. The abstract data type of the socket is given in Table 11.6.

<table>
<thead>
<tr>
<th>Sort</th>
<th>Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imports</td>
<td>Data, Configuration, Event, Service, Data Channel (DC), Event Channel (EC), Configuration Channel (CC), Service Collection (SC), Event Collection (EVC)</td>
</tr>
<tr>
<td>Operations</td>
<td>Initialize (DC, EC, CC) \rightarrow Socket</td>
</tr>
<tr>
<td></td>
<td>Add (Socket, Plug) \rightarrow Socket</td>
</tr>
<tr>
<td></td>
<td>Remove (Socket, Integer) \rightarrow Socket</td>
</tr>
<tr>
<td></td>
<td>Count (Socket) \rightarrow Integer</td>
</tr>
<tr>
<td></td>
<td>IsRegistered (Socket, Integer) \rightarrow Boolean</td>
</tr>
<tr>
<td></td>
<td>Set (Socket, Integer, Configuration) \rightarrow Boolean</td>
</tr>
<tr>
<td></td>
<td>Send (Socket, Integer, Data) \rightarrow Boolean</td>
</tr>
<tr>
<td></td>
<td>Receive (Socket, Integer) \rightarrow Data</td>
</tr>
<tr>
<td></td>
<td>Discover (Socket, Integer) \rightarrow SC</td>
</tr>
<tr>
<td></td>
<td>Detect (Socket) \rightarrow SC</td>
</tr>
<tr>
<td></td>
<td>GetService (SC, Integer) \rightarrow Service</td>
</tr>
<tr>
<td></td>
<td>GetCount (SC) \rightarrow Integer</td>
</tr>
<tr>
<td></td>
<td>Provide (Socket, Integer, Service) \rightarrow Boolean</td>
</tr>
<tr>
<td></td>
<td>Consume (Socket, Integer, Service) \rightarrow Boolean</td>
</tr>
<tr>
<td></td>
<td>Identify (Socket, Integer) \rightarrow EVC</td>
</tr>
<tr>
<td></td>
<td>Generate (Socket) \rightarrow EVC</td>
</tr>
<tr>
<td></td>
<td>Find (EVC, Integer) \rightarrow Event</td>
</tr>
<tr>
<td></td>
<td>GetSize (EVC) \rightarrow Integer</td>
</tr>
<tr>
<td></td>
<td>Listen (Socket, Integer, Event) \rightarrow Boolean</td>
</tr>
<tr>
<td></td>
<td>Dispatch (Socket, Integer, Event) \rightarrow Boolean</td>
</tr>
<tr>
<td></td>
<td>Handle (Socket, Event) \rightarrow Boolean</td>
</tr>
</tbody>
</table>
Axioms

- Count (Initialize (DC, EC, CC)) = 0
- Count (Add (Socket, Plug)) = Count (Socket) + 1
- Count (Remove (Socket, Integer)) = Count (Socket) – 1, if Count (Socket) > 0
- IsRegistered (Socket, Integer) = False, if Integer > Count (Socket)
- Set (Socket, Integer, Configuration) = False, if IsRegistered (Socket, Integer) = False
- Send (Socket, Integer, Data) = False, if IsRegistered (Socket, Integer) = False
- Receive (Socket, Integer) = Undefined, if IsRegistered (Socket, Integer) = False
- Discover (Socket, Integer) = Undefined, if IsRegistered (Socket, Integer) = False
- GetEvent (SC, Integer) = Undefined, if Integer > GetCount (SC)
- Provide (Socket, Integer, Service) = False, if IsRegistered (Socket, Integer) = False
- Consume (Socket, Integer, Service) = False, if IsRegistered (Socket, Integer) = False
- Identify (Socket, Integer) = Undefined, if IsRegistered (Socket, Integer) = False
- Find (EVC, Integer) = Undefined, if Integer > GetSize (EVC)
- Listen (Socket, Integer, Event) = False, if IsRegistered (Socket, Integer) = False
- Dispatch (Socket, Integer, Event) = False, if IsRegistered (Socket, Integer) = False

<table>
<thead>
<tr>
<th>TABLE 11.6: SOCKET EXPECTING SERVICES</th>
</tr>
</thead>
</table>

The incompatibility between the plug and socket is due to the use of different constructs for providing the desired extension capabilities. While the plug provides the desired capabilities by means of function invocation, the socket expects the desired extension capabilities to be provided by means of services. This is further evident when the Sorts that are imported are compared. While the plug imports the Function and Function collection sorts, the socket imports the Service and Service collection sorts. The various operations that these two components provide, further bring-forth the incompatibility. The plug supports the operations: Discover, Enumerate, Locate, Count, Invoke, RemoteInvoke on the Function/Function collection sorts, while the socket supports the operations: Discover, Detect, GetService, GetCount, Provide and Consume on the Service/Service collection sorts.

11.9 Summary

This chapter introduces and elaborates the generic plug and play component model. The principles governing the Plug and Play based architectures were put forth. Understanding the key architectural principles of this experiment helped identify the underlying plug and play architectural abstractions. An overview of some of the implementation constructs that are useful for Plug and Play based architectures was provided. Based on all this, the generic Plug and Play component model was derived. This generic model was instantiated to illustrate the plug and play architectures of a system comprising 4 different components. A possible scenario expressing an approach to identify the incompatibility between the plug and socket was also illustrated.