Chapter 8: Experiment 4 – Visual Process Programming Platform

This chapter introduces and elaborates the architectural abstractions that were utilized in the experimental system titled Visual Process Programming Platform. While there exist a significant number of architectural elements and design patterns in this system’s architecture, the focus of this discussion is on those that are potentially useful as Plug and Play constructs. The purpose of this platform is to capture all relevant information that is necessary to express a process in a form suitable for enactment and supports its execution by allocation of activities to appropriate agents and tracking their performance. To facilitate this, the platform provides the capability to encode, analyse and orchestrate processes that are expressed visually using different process modelling notations. Accordingly, the platform adopts an extensible architecture so that process modellers and process execution engines can be developed independently and then plugged into the platform on demand. The reasons for adopting an extensible architecture include the need to extend the functionality required by the agents to perform the activities, the ability to utilize a wide variety of software systems which will aid in performing the activity, the complexity of the process domain, and finally to allow agents adoption of different methods for performing their allocated activities.

In the context of Plug and Play, the objective of this experimental system is to “support plug and play based extensibility on two dimensions”. Wherein, one type of plugged components provide the ability to express the process using different process modelling notations, the other type of plugged components provide the ability to support the enactment of the process by using different execution engines. This is achieved by providing process programming (in the first type of components) and execution (in the second type of components) services as part of the plugged components. Accordingly, the hypothesis for this experiment is: “If components are plug and playable, then two different types of components can be plugged without functionally overlapping each other”. The core functionality of the platform are: a) support for a wide variety of modellers in order to encode processes and b) support for a wide variety of execution engines in order to execute processes. The methodology adopted to validate the hypothesis is to treat each of the modellers as a pluggable component and each of the execution engines as another type of pluggable components. Accordingly, the goal of this experimental system is to “enable Plug and Play of process modellers and process execution engines”.

This chapter is organized as follows: a) The section on the problem space provides an overview of the visual process programming problem, b) The section on the solution space provides a possible solution that is Plug and Play compliant, c) The section on the requirements lists down the stakeholder concerns,
d) The section on the quality characteristics lists down the functional and non-functional characteristics of the system, e) The section on decomposition brings forth the underlying component framework and supporting run-time, f) The next 8 sections provide an overview of the various components and their composition, and g) The last section discusses about the Plug and Play architectural abstractions of this experiment. It uses the architecture description template presented in Annex A to express the architecture description of the visual process programming platform.

8.1 The Problem Space

The complexity, scale and size of industrial processes are increasingly driving the need to improve the productivity of people involved in process modelling and orchestration in enterprises [111-120]. As more and more processes are digitized, it has become important to improve the rigor in which processes are expressed so that machines can aid humans in process orchestration [111-120]. Usually, a process can be recognized as a set of tasks performed in an organized manner under a set of constraints to achieve a particular goal. It gives the ability to role players to discharge their responsibilities in a systematic structured way. Organizations utilize processes to deliver quality offerings [111-120]. By controlling the quality of their process orchestration, organizations control the quality of their offerings [111-120]. The various tasks that are defined as part of the processes have to be orchestrated in a specific order by role playing people or by role playing machines to achieve the desired quality offerings.

Process modelling in such cases, involves understanding the process outcomes; the various work products that emanate from the execution of the tasks by these role-playing actors; the tools, techniques and methodologies that are utilized in performing these tasks; the organization resources that are necessary for performing the tasks; the constraints within which the role playing actors should operate and expressing them by utilizing an appropriate modelling notation that will enable process modellers to capture all the relevant information pertaining to the process-of-interest [111-120]. Process models serves to provide the means to develop the logic of what an organization does to deliver quality organizations. Process modelling constructs serve as the building blocks for defining this logic. On the other hand, process orchestration involves understanding the competencies of the various role-players, the schedules within which to operate, the operational resources that need to be utilized, the availability of these operational resources, the constraints in terms of policy guidelines, principles, directives and rules within which the process can be orchestrated, and the operating environment in which the orchestration is performed. Process orchestration generates the inputs and outputs that an observer outside the organization sees over some observation period. The ordering of tasks contribute to the
transformation of the inputs into incremental work products which are finally transformed into the process outcomes with desired quality characteristics.

To support these capabilities, the existing enterprise process solutions, enforce strict checks and balances, so as to assert that the necessary information pertaining to the tasks are available beforehand before the process orchestration is taken up [117]. Often, in practice, this is not really the case as situations demand that processes are unfurled dynamically. There are not many end-to-end systems that are utilized by enterprises to support such dynamic unfurling of processes. In most cases, a programmatic intervention is utilized (BPEL, WebSphere, WebLogic and so on) to achieve this functionality [117]. This intervention is not an ideal solution in such cases but enterprises have accepted it because that is the best that exists. Often, there exist islands of information systems or services which are connected by using automation agents and people resources to support the process orchestration. Moreover, not all process modelling notations can be orchestrated by the use of information systems, notations like BPMN are predominantly used for describing processes, using a rich set of visual notations, and the process flow semantics. On the other hand, notations like BPEL are predominantly used for integrating processes with services, in order to support process orchestration. In other words, some process notations are useful during the modelling stage while others are useful during the orchestration stage and to support both modelling and orchestration, there should be a way to seamlessly integrate processes expressed in both these notations. Another issue is that there is no one-size-fits-all solution for supporting process solutions, each of the available solutions have been found to be useful in a particular context and needs to be modified to be utilized in other situation contexts.

8.2 The Solution Space

A process denotes the set of all activities that are carried out in the context of achieving the desired outcomes. A process model is a representation of the process and contains the description of the activities to be executed, tools, resources, capabilities to be used, work products to be developed, inputs, outputs and constraints that affect the execution of the process. A key concern in this regard is the design of the process infrastructure for supporting process modelling and process execution.

The usefulness of process modelling is often limited by the fact that there are many approaches to construct the process based on the intrinsic complexity of the desired outcomes and their quality characteristics. Modelling notations like BPMN, BPEL and OPF are domain agnostic and as a result it has been very difficult to express everything about the way tasks are performed in different domains effectively using these notations. Traditionally, responsive visual programming have helped designers to
express their program logic better by adopting a) fewer concepts to express the program, b) standardized programming process, c) explicit relationships, and d) immediate feedback. This approach has enabled development environments to express more information that is relevant to the situation context than that has been possible using traditional programming languages. In this experiment, the usage of visual programming as the underlying paradigm for expressing processes is discussed.

Every process modelling notation has operational semantics associated with each of the individual constructs that is supported by the notation. Some constructs require human involvement in performing the tasks while others can be performed by automation agents by means of services or APIs or function invocation. Accordingly, while modelling the processes using these notations, tasks that are performed by humans can be tagged with information that are human interpretable, and tasks that are performed by automation agents can be tagged with information that are structure and machine interpretable. This schema is a useful way of expressing characteristics of the process steps. During process orchestration, information pertaining to the task to be performed, in terms of artefacts, resources, timelines, budgets, are necessary for triggering the actions. Additionally, audit information like time, cost, and performance indicators are necessary for further analysis of the orchestrated process. All these can be tagged as placeholders for every task which are then filled up when the process is orchestrated.

In this setting, the goals of the visual process programming platform are:

- Create a process infrastructure for supporting process modelling and process execution.
- Support different process modelling and execution approaches and associated semantics.
- Support a wide variety of operational semantics and relevant constructs that will enable the modelled tasks to be performed either by humans or by machines.
- Support dynamic unfurling of tasks into a full process when that particular task is taken up for execution.
- Provide placeholders for information pertaining to the metrics and other indicators that are captured when the process is orchestrated.
- Embed work products and associated templates as part of the tasks so that when the tasks are performed by a role player, the work products are built automatically.

### 8.3 Requirements for the Visual Process Programming Platform

i) Process designers should be able to select a specific modelling notation from a collection and express their process using this notation.
ii) Process designers should have the ability to modify the modelling notations according to their intended purpose of usage of the process.

iii) When the underlying modelling notation is changed, the process model should be able to accommodate the enhancements.

iv) Process designers should be able to express a process using multiple modelling notations.

v) Process designers should be able to exchange process steps and process fragments across different processes.

vi) Process designers should be able to work on multiple process models at the same time.

vii) Process designers should be able to expand the collection of modelling notations available at their disposal.

viii) Process designers should be able to save the state of their models and resume modelling at a later point in time.

ix) Process designers should be able to rework their models from any point of their model formulation. They should be able to reuse their models across different modelling instances.

x) Process designers should be able to augment their process models with additional information that are relevant to the situation context.

xi) Process designers should be able to embed work products and templates as part of their process definitions.

xii) Process designers should be able to augment the tasks expressed in the process models with additional information pertaining to the role players who will perform the actions.

xiii) Process designers should be able to define abstract role-players and resources for their tasks as part of the process models.

xiv) The platform should support a wide variety of operational semantics pertaining to the domain related tasks that are encoded in the process.

xv) The platform should allow addition and modification of the operational semantics corresponding to an encoded task.

xvi) Audit trials corresponding to the orchestration of the process should be automatically logged.

xvii) Users should be able to bind Resources, Work products and Role players during execution.

xviii) Users should be able to unfurl tasks as full blown processes during execution.
8.4 *Quality Characteristics of Visual Process Programming Platform*

Quality characteristics of a system are a set of essential and distinguishing attributes that have a pragmatic interpretation of the system’s inferiority or superiority. By controlling the quality characteristics, it is possible to ensure that the system delivers desired value to its stakeholders [100-102] [132-133]. For this discussion purposes, the author utilized ISO 25010 [103] as the reference quality model. The subsequent sections list down the quality characteristics of the Visual process programming platform.

1. Functionality:
   a. Suitability: Aids process designers in expressing at least 80% of the processes that they encounter.
   b. Suitability: At least one operational semantic can be used to perform a task in the model.
   c. Suitability: At least one modelling notation can be used by the process designer to model their process.
   d. Suitability: Aids organizations in orchestrating at least 80% of the modelled processes.
   e. Accurateness: Process designers should be able to accurately represent the model of their process along with sufficient detail.
   f. Interoperability: Process designers can transfer process steps and process fragments to different modelling notations.
   g. Interoperability: An operational semantic should be relevant to more than one task.
   h. Compliance: Complies with all the modelling constraints advocated by the modelling notation.
   i. Compliances: Complies with all the constraints pertaining to an operational semantic.
   j. Security: Process designers can be assured that the information expressed in the process model is confidential.
   k. Security: During process orchestration, execution information is shared with the right role players.
   l. Security: Uses open format for storing information in storage media while modelling and orchestrating.

2. Reliability:
   a. Maturity: Modelling notations are mature enough to help address 80% of the modelling needs of process designers.
b. Maturity: Operational semantics are mature enough to help address at least 80% of the execution needs during process orchestration.

c. Fault Tolerance: Any issues that arise due to implementation of a modelling notation should be handled gracefully.

d. Fault Tolerance: Any issues that arise due to execution of a task should be handled gracefully.

e. Recoverability: On exceptions, uses rules and conditions to rollback or abort tasks (50%)

f. Recoverability: Store and Retrieve models and model workspaces from a persistent storage media.

g. Recoverability: Store and retrieve information pertaining to task execution from a persistent storage media.

3. Usability:

a. Understand ability: Process designers can comprehend and use the modelling notation (100%)

b. Understand ability: Role players can comprehend the actions that are allocated to them (100%)

c. Learnability: Guides through the steps of using the modelling platform (50% guidance)

d. Learnability: Guides through the steps of performing the allocated tasks (100% guidance)

e. Operability: Uses familiar interfaces for interacting with the User (80% compatibility)

f. Operability: Undo/Redo last few actions performed by the user on the Platform

g. Operability: Uses familiar tools for performing tasks during process orchestration.

4. Efficiency:

a. Time Behaviour: Process designers should be able to model their processes within a predefined time frame. The platform should not be processing forever.

b. Time Behaviour: During process orchestration, a task is executed within a predefined time frame. The platform does not hold a task forever.

c. Resource Behaviour: Platform should use minimal resources to satisfy the modelling needs of the process designer.

d. Resource Behaviour: Platform uses available resources during task execution. If sufficient resources are not available then appropriate exceptions are triggered.

5. Maintainability:

a. Analysability: Process designers can keep track of progress always (100%)

b. Analysability: Process execution progress can be tracked always (100%)
c. Changeability: Process designers can change their models and modelling notation on demand (100%)

d. Changeability: Resources, Work products, Templates and timelines can be modified on demand.

e. Stability: Absorbs 80% of perturbations introduced by the working environment.

f. Testability: System performance and its use to model a process is reproducible (100%)

g. Testability: Process performance on the same set of inputs and constraints is reproducible (100%)

6. Portability:

   a. Adaptability: Adapts to change in the working environment (100%)

   b. Install ability: It should be easy to expand the collection of modelling notations that are available in the Platform (90%)

   c. Install ability: It should be easy to add more role-players and agents during process orchestration.

   d. Conformance: Conforms to Modelling constraints (100% compliance)

   e. Conformance: Conforms to organizational constraints (100% compliance)

   f. Replace ability: Process designers can replace any of their Models and modelling notations (100%)

   g. Replace ability: Operational semantics, resources, role players can be replaced (100%)

8.5 Decomposition of the Visual Process Programming Platform

The domain of processes utilizes visual representation as the medium of expression wherein only the essential details pertaining to the process is displayed and the rest of the information is obfuscated [111-120]. In order to create a process-centred environment, it becomes obvious that the visual programming paradigm should be supported. Process programming, in this case, implies the design of processes that fit a computing environment and can be tested, debugged and modified much like computer programs [117]. There are three critical capabilities that needs to be supported for this purpose [117]. The first capability is process modelling, wherein the ability to create a representation of an executable process in a form that is suitable for orchestration by the machine is supported. The second capability is process enactment, wherein the ability to allocate tasks to role players (humans or machines) and monitoring the task performance for completion is supported. The third capability is process debugging, wherein the
ability to simulate process execution and analyse the potential issues is supported. Collectively, these three capabilities dictate the structure and behaviour of the visual process programming platform.

In this platform, process modelling is considered as the act of creating a representation of a process in a form that is suitable for execution. Amongst the different process modelling approaches, the one that is suitable for supporting process enactment involves utilizing a process programming language and associated visual notations to express the process. Accordingly, process modelling involves: a) Identifying the activities that needs to be encoded, b) Identifying the inputs, outputs, conditions and constraints for each of these activities, c) Identifying the necessary human, machine and organizational resources that can aid in enactment, d) Identifying the work products that will be created, e) Identifying the order in which these activities are performed, and f) Identifying the flow of information between these activities.

In this platform, process enactment is considered as the act of interpreting and executing a process in a process centric environment. It provides mechanisms for agent interaction, agent guidance, activity allocation, activity tracking, outcome management, resource management and facilitates traceability between process decisions, rationale and the resulting outcomes. Accordingly, process enactment involves: a) Identifying the execution semantics corresponding to different activities, b) Associating supplemental information to the activities in order to support its execution, c) allocating activities of a process model to different agents, d) supporting the performance of these activities, e) managing the performance of these activities, f) managing the resources, information needs, constraints and artefacts involved in performing these activities, g) monitoring the work products that are created, h) tracking the state of the process execution and i) gathering information about the performance of the activities.

In this platform, process debugging is considered as the act of analysing the execution of a process in a simulated test environment. It provides mechanisms for identifying process “bugs”, simulating agents, simulating activity performance, execution state trace, process recovery, and failure modes. This is achieved by using the process execution capabilities to execute a process in this test environment and collect different traces. Accordingly, the visual process programming platform can be decomposed into the top level components: a) Process modelling framework that supports the ability to express different processes, b) Process modelling framework run-time that supports the modelling framework, c) Modeller container which manage a collection of modellers, d) Process enactment framework that supports the ability to execute processes, e) Process enactment framework run-time that supports the enactment framework, f) Enactment container which manages the execution engines, g) Data Repository that supports the information needs of the modelling and enactment frameworks and h) Model Repository
that supports the management of the models used by the modelling and enactment frameworks. This decomposition is presented in Figure 8.1.

**Figure 8.1: Decomposition of the Visual Process Programming Platform**
8.6 Process Modelling Framework

The process modelling framework (PMF) present in the platform provides a set of generic capabilities, resources and interfaces for process modelling which serves as the foundation for the creation and assembly of process models.

![Figure 8.2: Decomposition of Process Modelling Framework]

The core capabilities provided by the framework are: a) Model rendering, b) Model storage and retrieval, c) Model traversal, and d) Model assembly. In order to develop the process model, it is necessary to understand the process design and encode it using the chosen modelling language that supports execution. Additional information like the inputs, outputs, constraints, work products, resources, artefacts, bindings and so on needs to be expressed using this language in order to have complete information about the process. Supplemental information pertaining to the execution of the process should be expressed using this language so that the process is ready for execution. As shown in figure 8.2, PMF uses the services provided by the modeller to create a representation of the process that conforms to the specific process language. It uses the services provided by the model assembler to string different process steps and process fragments together in the form of a process. It uses the services provided by the model renderer to generate different views of the process model. It uses the modelling services provided by the plugged modellers to provide support for the different processes. It uses the tree traversal services provided by the model traverser to navigate thru the process model hierarchy.
8.7 Process Modelling Framework Run-Time

Process modelling involves expressing a process using a process language and associated diagramming notation. Modularity and extensibility are design principles that facilitate extension of the process modelling capabilities of the platform by facilitating plug and play of the modellers. A good design practice in this situation, is to embed the necessary plug and play services in the run-time system. The objective of the run-time system is to provide necessary interfaces that support interaction with the modellers, sharing of information with the modellers, integration of the modellers with the platform and management of model life-cycle. In this setup, the bus design pattern can be used for supporting information sharing, the host-controller design pattern can be used to facilitate component interactions, the plug and socket design pattern can be used to dynamically plug and play process modellers at run-time, the event-driven design pattern, adopted by graphical user interface based operation systems can be used in PMFR to handle and process the various events that are triggered by both the platform as well as the modellers. The services design pattern, adopted by service-oriented-architectures can be used in PMFR to handle consumption and provision of services that are of interest to the other components of the process modelling platform.

![Figure 8.3: Software Decomposition of Process Modeller Framework Run-Time](image-url)
As shown in figure 8.3, the modelling framework run-time is a composition of: a) Modeller Socket which serves as the extension point for a modeller, b) Host Controller which provides resources, capabilities and services to the modellers, c) Model Manager which provides resources, capabilities and services for manipulating and transforming Models, d) Event Handler which serves as the placeholder for invoking services/functionalities from various modellers and processing their results in response to the various events triggered by the modelling platform, e) Service manager which provides the mechanism for providing services that can be utilized by the other components and for consuming services that are provided by the other components.

8.8 Process Execution Framework

Process execution is the process by which the activities identified in the process model is performed by a human or automation agent. The effect of these activities depends on the configuration of the activities as defined in the process model and the supplemental execution information that is provided by the platform. It also depends on the semantics of the process modelling language. During process execution, the activities are performed in a prescribed order that is encoded in the process model. Process execution typically starts at the “start” activity of the process. When each of these activities are taken up, a set of actions is allocated to the different agents who then go about and perform these actions.

The core capabilities provided by the framework are: a) Activity allocation and sequencing, b) Activity performance monitoring, c) Activity tracking, d) Work products tracking, e) Information management, and f) information flow management. Additional information like actors, artefacts, work product repository should be provided in order to execute the process. The Process execution framework, as shown in Figure 8.4, is designed to support the process models that are developed based on the underlying process language. During execution, the state of the execution and the performance indicators are saved in persistent storage as part of the process data of that specific execution instance. This information is useful in cases where the execution is interrupted and is restarted at a later point in time.

As shown in figure 8.4, Process Execution Framework uses the services provided by the executor to execute the specified process. It uses the services provided by the information manager to manage data used during the execution of the process. It uses the services provided by the flow manager to pass information between different activities that are part of the process. It uses the services provided by the activity monitor to support, track and gather information about how the activity is performed. It uses the services of the activity allocator to assign activities to different agents who are responsible for the
successful orchestration of the process. The transaction and process data to be utilized for the execution is initialized and populated with the data that is relevant for this particular execution instance.

**Figure 8.4: Decomposition of Process Execution Framework**

### 8.9 Process Execution Framework Run-Time

Process execution refers to the execution of the process as expressed by its process model by means of appropriate agents, resources and information. To support a variety of process languages that provide execution support, the platform facilitates extensions of the process execution capabilities by facilitating plug and play of the execution engines. A good design practice in this situation, is to embed the necessary plug and play services in the run-time system. The objective of the run-time system is to provide necessary interfaces that support interaction with the execution engines, management of data necessary for execution, tracking the progress of the execution and management of process execution life-cycle.

In this setup, the bus design pattern can be used for supporting information sharing, the host-controller design pattern can be used to facilitate component interactions, the plug and socket design pattern can be used to dynamically plug and play execution engines at run-time, the event-driven design pattern can be used to listen and dispatch the various events that are triggered by the platform and the execution engines. The services design pattern, adopted by service-oriented-architectures can be used to handle consumption and provision of services.

As shown in figure 8.5, the execution framework run-time is a composition of: a) Execution Socket which serves as the extension point for an execution engine, b) Host Controller which serves as the node that provides resources, capabilities and services to the execution engines, c) Data Manager which
provides resources, capabilities and services for managing execution data, d) Model manager which provides the necessary services for interpreting the process models, e) Event Dispatcher which serves as the placeholder for listening to different events and dispatching them to appropriate handlers, and f) Service manager which provides the mechanism for supporting service consumption and provisioning.

**FIGURE 8.5: SOFTWARE DECOMPOSITION OF PROCESS EXECUTION FRAMEWORK RUNTIME**

### 8.10 Process Modeller

In this platform, process modellers are used to build a representation of the selected processes for the purposes of execution. Process languages and an associated diagramming notation is used to represent these processes. The level of details expressed in a process model depends on the complexity of the execution (human agents and machine agents). Generally, the sequence of activities that are to be done within a timeframe by an agent while utilizing a specific set of resources while satisfying a set of constraints is expressed as a process model. It is developed by stringing together a series of activities in a specific sequence with the objective of delivering quality outcomes.
Each activity is interdependent on another activity (output from one becomes the input for another). As a result, many of these activities overlap on different factors and it is necessary to develop an appropriate process model by taking these concerns into consideration. The extent of the success of modelling depends on the ability of the process language to capture the essence of the process. There are five main characteristics of a process that a modeller captures as part of the model: a) Activity to be performed, b) Semantics of the activity, c) Sequence or flow of activities in relation to each other, d) information that is necessary for performing the activity, e) Specification of the work products that the activity produces, f) inputs and constraints, g) Supplemental information that aids in process execution and h) Information flows between the different activities.

Accordingly, the process modeller, as shown in Figure 8.6, is a decomposition of: a) Activity modeller for expressing the process steps to be performed by a role player, b) Information modeller for expressing the supplementary information pertaining to each process step, c) Resource modeller for expressing the
resources and capabilities necessary for performing the process step, d) Work product modeller for expressing the outcomes and outputs that emanate from the performance of each process step, e) Input modeller for expressing the inputs necessary for performing the process step, f) Constraint modeller for expressing the constraints that affect the performance of a process step, g) Sequence modeller for expressing the order in which the process execution should occur and h) Operations modeller for expressing any other supplemental information needed for process execution. The different modellers use the syntax and notations provided by the underlying process language to express the different process elements. In cases where the underlying process language does not have appropriate constructs corresponding to the process element then those process elements are not captured as part of the model.

8.11 Process Modeller Run-Time

As discussed in section 8.10, the process modeller aids in capturing different levels of detail of a process and uses different process languages and associated notations for this purpose. It utilizes the just-in-time services provided by the run-time for supporting data handling, event handling, dynamically plug and playing modellers and for service invocation. The runtime is designed to handle discovery of socket interfaces, connection to sockets, information exchange with sockets, event listening and dispatching to sockets and disconnection from sockets. It provides a gateway to the runtime environment which contains active components that can be interacted with during the execution of the platform. Once a modeller is plugged into the modelling framework, the run-time serves as an abstraction layer that extends the capabilities of the framework by considering these modellers as constituent elements of the framework.

![Diagram of Process Modeller Run-Time](image)

**Figure 8.7: Software Decomposition of Process Modeller Run-Time**
As a result, the modeller component is treated as par with a normal component that is part of the modelling framework. To facilitate this, the run-time utilizes the interface services offered by the interface manager to dynamically attach to and detach from the modeller socket. It utilizes the capabilities of the service manager to consume services offered by the modelling framework and to provide services to the modelling framework. It uses the capabilities of the model manager to exchange information pertaining to the process models that are manipulated by the modeller with the modelling platform. It utilizes the eventing services provided by the event dispatcher to listen and dispatch the events that are triggered by the modeller as well as the modelling framework. This decomposition of the run-time is presented in figure 8.7.

8.12 Execution Engine

In this platform, execution engines are used to support the creation and management of the execution of the process instances. Every execution engine supports a set of operational constructs that are believed to be an appropriate interpretation of the elements of a process language. When executing the process that is expressed using this process language, each activity in the process is associated to one or more of these operational constructs and an appropriate action invoked. The outcome of this action results in the creation of one or more work-products, that have been identified as part of the process definition, which are then stored in the data repository for further processing. Since each action runs within its own execution space, it becomes possible to execute concurrent actions at the same time.

During process execution, the activities are executed in the prescribed order that is encoded in the process model. Each activity is taken up when the conditions that is necessary for performing the related actions are satisfied. When all the activities pertaining to a specific process instance is executed then it signifies that the process has been executed and the data repository will contain the process outcome. The extent of the success of the execution depends on the process definition, the process language used, the operational constructs that were invoked and the ability to successfully complete executing all the activities that lies in the order of the process. The main characteristics that the execution supports are: a) Process interpretation, b) Process instantiation, c) Resources and agents binding, d) Activity allocation, e) Activity performance by action invocation, f) Activity tracking, g) Work product house-keeping, h) Information management, i) Information flow management, j) Activity flow management, k) Process outcome management, and l) Process performance management.
Accordingly, the execution engine, as shown in Figure 8.8, is a decomposition of: a) Process interpreter for instantiating the process for a specific situation and breaking it down into activities and control flows, b) Resource manager for provisioning desired resources on demand, c) Execution Agent (can be human agents or automation agents) for performing the related actions, d) Activity allocator for assigning the activity to different agents, e) Activity tracker for tracking the performance of the related actions, f) Information manager for managing the information needs of the various agents, g) Work product manager for managing the work-products that are created by the different agents and h) Performance manager for managing the performance of the activity by the execution agents. The process interpreter and the activity allocator collectively ensure that the constraints necessary for executing the activity are met before the activity allocation.

**Figure 8.8: Software Decomposition of Execution Engine**
8.13 Execution Engine Run-Time

As discussed in section 8.12, the execution engine aids in instantiating a process for a specification situation and executing it. It uses different operational constructs for this purpose. It utilizes the just-in-time services provided by the run-time for supporting data handling, event handling, dynamically plug and playing execution engines and for service invocation. The runtime is designed to handle discovery of socket interfaces, connection to sockets, information exchange with sockets, event listening and dispatching to sockets and disconnection from sockets. It provides a gateway to the runtime environment which contains active components that can be interacted with during the execution of the process. Once an execution engine is plugged into the execution framework, the run-time serves as an abstraction layer that extends the capabilities of the framework by considering these execution engines as constituent elements of the framework.

As a result, the execution engine is treated as par with a normal component that is part of the execution framework. To facilitate this, the run-time utilizes the interface services offered by the interface manager to dynamically attach to and detach from the execution socket. It utilizes the capabilities of the service manager to consume services offered by the execution framework and to provide services to the
execution framework. It uses the capabilities of the specification manager to exchange information pertaining to the process models that are to be instantiated for a situation and executed. It utilizes the eventing services provided by the event dispatcher to listen and dispatch the events that are triggered by the execution engine as well as the execution framework. It uses the data management services provided by the Data manager to exchange execution data between the execution engine and the execution framework. This decomposition of the run-time is presented in figure 8.9.

8.14 Discussions

The experimental system addresses its primary goal of enabling plug and play of process modellers and process execution engines by:

a) Encapsulating process modelling functionality by encoding them into the modeller component.
b) Encapsulating process execution functionality by encoding them into the execution engine component.
c) Binding execution data and process performance indicators to the process data during enactment.
d) Supporting two different types of plugs and sockets (one for modellers and another for execution).
e) Encoding the plug and play interface mechanism as socket and compatible plugs.
f) Encapsulating the plug capability as part of the modeller run-time and execution engine run-time components.
g) Encapsulating the socket capability as part of the component framework run-time component.
h) Dispatching the events to the appropriate handler by using the plug & socket interface.
i) Providing the services to the appropriate consumer by using the plug & socket interface.

The critical elements events, services, process models and execution data that are required for proper functioning of the visual process programming platform have been made accessible through the plug and socket interface. As discussed in sections 8.6 and section 8.8, it supports two component frameworks that support two different types of plugged components (modellers and execution engines). It can then be observed that “Plug and Play architectures are polymorphic”. Accordingly, the hypothesis “If components are plug and playable, then two different types of components can be plugged without functionally overlapping each other” was verified. Further, the underlying principles and constraints that are discussed in the subsequent sections support extensibility along two dimensions.
8.14.1 Principle of compatible support elements

According to Maier [51], the architecture of the supporting element should fit into the system that it supports. In the case of the visual process programming platform, the plug interface of the modeller and the socket interface of the programming framework are designed such that only compatible modellers can utilize this interface. Similarly, the plug interface of the execution engine and the socket interface of the execution framework are designed such that only compatible execution engines can utilize this interface.

8.14.2 Principle of grouping and separation

According to Maier [51], while partitioning a system, it is necessary to group those elements that are strongly related to each other and separate elements that are unrelated. In the case of the visual process programming platform, all the components that are essential for supporting the modelling functionality have been grouped together and co-exist as part of the modelling framework and all the components that are essential for supporting the execution functionality have been grouped together and co-exist as part of the execution framework. Extensibility of the modelling framework is provisioned by plugging in modellers and extensibility of the execution framework is provisioned by plugging in execution engines. These two entities use the process model as the common ground for being part of the programming platform.

8.14.3 Principle of separation of concerns

According to Maier [51], while partitioning a system, it is necessary to leave the specialities to the specialist. Accordingly in the visual process programming platform, the process programmers (modellers) have been separated from the process executors (execution engines) in terms of the interface mechanisms, supporting services, underlying technologies, supporting events and also the framework into which they are plugged in. The platform is architected in such a way that all the modelling concerns are handled only by the modelling framework and all the enactment concerns are handled only by the execution framework.

8.14.4 Principle of minimal communication

According to Maier [51], while choosing the sub-systems, it is necessary to define them in such a way that they are independent to each other and there is minimal communication between them. Accordingly, in the visual process programming platform, the process model is the single collection of
information that is exchanged between the process modelling framework and the process execution framework. For all practical purposes, no other interactions happens between these two sub-systems.

8.14.5 Principle of independent partitioning and low complexity

According to Maier [51], while partitioning a system, it is necessary to choose the elements so that they are as independent as possible; that is elements of low external complexity and high internal complexity. In the case of the visual process programming platform, the process modellers are independent of each other and can serve as the extensible component. Similarly, the process execution engines are independent of each other and can serve as another extensible component.

8.14.6 Principle of accommodating change

According to Maier [51], in large systems, evolution is a process of ingress and egress. Accordingly, evolution in the case of the visual process programming platform is made possible by the ability to plug a modeller; to plug an upgraded or new modeller; and to unplug a modeller that has outlived its expectations. Evolution is also possible by the ability to plug an execution engine; to plug an upgraded or new execution engine; and to unplug an execution engine that is no longer useful. Further evolution is possible by modifying the information necessary for execution; by upgrading or providing new information; and removing information is no longer deemed useful.

8.15 The Plug and Play Architectural Abstractions

This section introduces and elaborates the plug and play architectural abstractions of the visual process programming platform. While many of these abstractions were discussed in earlier experiments, their capabilities have improved at the end of this experiment, and are expressed as such. These abstractions 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 visual process programming platform”. These abstractions are:

8.15.1 Plug

A plug comprises of a collection of software resources that enables the plugged component to be executed on an underlying computing system. It serves as the gateway for the events, data and services that the plugged component manipulate as part of its computation and allows two way communication between the component framework and the plugged components. The plug facilitates the
reconfiguration of the plugged component based on the configuration information that it receives from
the socket. The plug encodes the interface mechanism for connecting to a compatible socket thereby
enabling the plugged component to function as an external component of a component framework.

8.15.2 Socket

A socket is a handle that allows the plugged components to utilize the data, events and services that
are exposed by the component framework. It facilitates two way communication of events, data and
service invocation between the plug and the component framework. The socket provides the plug with
configuration information based on the changes requested by the component framework. As an
interface, the socket performs a type checking for compatible plugs to validate the connection and
disconnects those that fail the validation.

8.15.3 Component Framework

A component framework defines specific interaction and composition standards that should be
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 in terms of the services, service operations, events, and other metadata
that it should support. It handles events raised by all of its constituent elements (including the plugged
components) as well as dispatches events for processing by these constituent elements. It consumes
services provided by all its constituent elements as well as provides services required by them. It
customizes the plugged component by utilizing a pre-defined set of configuration elements.

8.15.4 Plugged Component

A plugged component extends and adds functionality to the component framework thereby enabling
the evolution of the underlying system. It utilizes the run-time services provided by the plug to attach
itself to the component framework. It processes various events that it receives as well as dispatches some
of these events to the component framework. It consumes services offered by the component framework
as well as provides services to it. It re-configures itself based on the configuration information received
through the plug.
**8.15.5 Event**

An event is an encapsulated set of software actions that are triggered during component execution, along with a set of data that capture the computational state under which the event was triggered. Such events are triggered when a set of pre-defined execution conditions are met. They can be triggered inside the component framework and dispatched to the plugged component for processing or triggered inside the plugged component and dispatched to the framework for processing.

**8.15.6 Service**

A service is an encapsulated set of software 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.

**8.15.7 Configuration**

A configuration is an encapsulated set of preferences supported by a component on a set of resources that is supported by the component. In the case of the plugged components, the configuration is the specific definition of the elements that affect the behaviour and performance of the plugged component as a whole. The configuration of the component can be modified by the services it provides at run-time thereby allowing different behaviour at different points in time.

**8.15.8 The Plug and Play Component Model**

The Plug and play component model is put together by considering the plug and play architectural abstractions that were discussed earlier. Accordingly, the constituent elements of The Plug and play component model are: a) The component framework which provides the composition semantics, the logical structure for facilitating components composition, underlying structure in terms of interfaces, resources and protocols for the composition of components playing specific roles, b) The component framework run-time which serves as a socket and provides the necessary functionality for supporting composition of components, communication between components, manipulation of components, management of components performance, invocation of functions and services supported by the components and transformation of data from one form to another, c) The Specification Repository which hosts the different configuration elements of the component framework, d) The Data Repository which hosts the data that is manipulated by the component framework and e) The components manager which manages the plugged components and their respective run-times. Figure 8.10 provides a decomposition
of this model. While the component framework hosts the functionalities and processes that are utilized by the end-users, its run-time provides functionality and resources needed by it, along with the ability to serve as a socket. It provides basic services necessary for various plugged components to work together, for raising and handling events, for consuming various services that are supported by the plugged component and for managing data flow between the plug and socket.

![Diagram of Plug and Play Component Model](image)

**Figure 8.10: Decomposition of Plug and Play Component Model**

### 8.16 Summary

This chapter introduces and elaborates the architectural abstractions that were utilized in the experimental system titled visual process programing platform. The approach to support plug and play based extensibility on two dimensions was illustrated by utilizing two different types of pluggable components that are attached and detached to the platform on demand. This has been exemplified by the use of two nearly independent component frameworks as discussed in sections 8.6 and 8.8. Accordingly, the hypothesis “**If components are plug and playable, then two different types of components can be plugged without functionally overlapping each other**” is verified. Understanding the key architectural principles of this experiment helped identify the underlying plug and play architectural abstractions. This was culled out and expressed as an abstract Plug and Play component model.