

An Executable Graph Component Database for Defining and Executing Computer Programs

Steven A. Gold

GraphLogic Inc.
sgold@graphlogic.com

David M. Baker

GraphLogic Inc.
dbaker@graphlogic.com

Abstract

An Executable Graph Component Database (EGCD) for defining and executing computer programs is described. An EGCD incorporates every aspect of a computing program, including process flow and logic, user interface definition, database schemas and persistent data. The major elements of an EGCD system are described, including the *Graph Component Database* (GCD), whose core components are nodes, edges and properties, the *Dynamic Graph Interpreter* (DGI), the *Application Controller Viewer* (ACV), the *Application Editor System* (AES) and the *Executable Graph Component Database-Application Program Interface* (EGCD-API). The PointDragon platform, a web-based Integrated Development Environment (IDE) and run-time environment, which is a practical, commercial realization of an EGCD system, is also described. The PointDragon platform incorporates an object oriented visual programming language, whose syntax and expressive power derives naturally from the underlying EGCD system. Other aspects of the PointDragon platform, which also derive from the EGCD system, are delineated including the IDE and run-time environment. The PointDragon Visual Programming Language (PVPL) and IDE are compared with traditional object-oriented programming and IDE's.

Categories and Subject Descriptors D.1.7 [Visual Programming]; D.2.6 [Programming Environments]; Graphical Environments, Integrated Environments; D.2.11 [Software Architectures]; Languages; D.3.2 [Programming Languages]; Very High-Level Languages.

General Terms Languages.

Keywords Executable graph component database; executable graph; executable database; visual programming.

1. Introduction

In the past fifty years, the computing industry has witnessed a billion fold increase in available computing power. Yet the fundamental paradigm for creating computer software has remained unchanged. The vast majority of today's software is still created using traditional programming languages that interact with a data repository. A fundamental split between mutable persistent data and immutable (or least difficult to change) program code is at the

core of the vast majority of computer software architectures developed in the last fifty years. Object oriented programming is an attempt to mitigate this split, by combining transient data and program code within objects; however, program code and persistent data still remain very distinct entities – the Java code of an object method is separate from the attribute values of a persistent object instance. Such differences are maintained throughout all general purpose software architectures – persistent data is stored in a database, which may be relational or object oriented – while the program (e.g., Java or C++ code) is typically stored in a code file. Although some database systems provide a way to store program code in a database, the data and program code are typically kept in separate structures and in very different forms in such systems.

In contrast, the Executable Graph Component Database (EGCD) [1,2,3,4] described herein stores all the process and computational logic, display properties, database schemas, metadata and instance data of an application in a single graph oriented structure. All application elements are stored in the Graph Component Database (GCD) as nodes, edges and properties. The EGCD system naturally synthesizes program and data, which eliminates the split between program and data found in the vast majority of general purpose computing software architectures.

A Dynamic Graph Interpreter (DGI) interprets a Graph Component Database (GCD) to run a program. A Graph Component Database contains a dynamic graph which is modified while the DGI executes the program. The nodes, edges, and properties of a GCD form the basic units of a computer program. This provides a fine-grained modular structure that can grow and be updated in an incremental fashion. The GCD can be changed at any time, including while the application is running. Changes are implemented by adding, updating or deleting nodes and edges or by changing their properties.

The GCD is not object-oriented. The basic elements are nodes, edges and properties, not objects. However, a language the GCD describes and executes can be object oriented. The PointDragon platform [5] is the first implementation of an EGCD based system. It utilizes the expressive power of EGCD to implement a codeless visual programming environment. PointDragon is a web-based development and run-time environment. It is a successful commercial realization of an EGCD system. The PointDragon platform is currently being used to run large-scale enterprise systems at biotech and pharmaceutical companies [6]. The PointDragon platform implements an object oriented visual programming language (PVPL) [7,8,9,10], whose syntax and expressive power derives naturally from the underlying EGCD system. All three elements of the PointDragon platform, the PVPL, the IDE and the run-time platform are naturally integrated by the EGCD.

The PointDragon platform and underlying EGCD system have been implemented using traditional object-oriented technology: Java and ObjectDB, an object-oriented database [11].

In this paper the major elements of an EGCD system are described, including: (1) the *Graph Component Database* (GCD); (2) the *Dynamic Graph Interpreter* (DGI); (3) the *Application Controller Viewer* (ACV), which is the component through which a user communicates with the DGI; (4) the *Application Editor System* (AES), which is the component that provides an external user interface for directly editing the nodes, edges and properties of a GCD in a manner and with a syntax similar to a computer language, typically a visual programming language; and (5) the *Executable Graph Component Database-Application Program Interface* (EGCD-API), which allows the EGCD to communicate with external computer systems.

2. The Executable Graph Component Database

2.1 Graph Component Database (GCD)

The Graph Component Database is a composite layered graph with two node types – composite and primitive. Nodes composed of other nodes are composite; whereas, nodes that do not contain other nodes are primitive. All nodes within a GCD are connected by edges that define the relationships between nodes. The relationships may include, but are not limited to: transformational, passive, control passing and relational. The GCD composite layered graph includes a set of directed graphs and a set of trees. Each node within the graph belongs to both a directed graph and a tree. The directed graph to which each node belongs defines its relationships with other nodes within that graph. The tree to which the node belongs defines the composition hierarchy to which it belongs. A node may belong to multiple directed graphs as well as multiple composition hierarchies.

The nodes and edges of a GCD are used to define both data and program logic structures of arbitrary complexity. They provide the common form and function that fuses every aspect of a computing program while retaining a common, fine-grained modular structure that can grow and be updated in an incremental fashion. In addition to defining static data and program logic structures, GCD nodes and edges define the dynamic state of a program's data and its logic flow as it is interpreted or executed. Graph Component Database applications are accessed, interpreted, modified, and run by the Dynamic Graph Interpreter (described below). The relationship between the GCD and DGI is depicted in Figure 1.

2.2 Dynamic Graph Interpreter (DGI)

The Dynamic Graph Interpreter runs an instance of an application by making transitions from one application state to another and displaying the application's state information via the Application Controller Viewer (described below) on an internet browser. This is equivalent to interpreting, executing, performing or running a computer program in the traditional sense. The DGI makes it possible to concurrently develop and modify an application while it is being run. In addition, it allows such development and modification to apply to either all subsequent runs of an application, a subset of subsequent runs, or simply to the current instance of the application being run. This inherent feature of the GCD and DGI facilitates the rapid development and long-term maintenance of application systems. User access to this dynamic development capability is tightly controlled in a production environment by the GCD-DGI multi-level security and access system. In addition to data entered or changed via terminals, the system also accepts

input data to application processes in various digital formats, including industry standards such as XML and web services.

2.3 Application Controller Viewer (ACV)

The Application Controller Viewer consists of an Application Controller and an Application Viewer. The Application Controller Viewer together with the GCD and DGI form the Dynamic Model View Controller application structure shown in Figure 2. This dynamic application structure manifests running instances of EGCD applications. The Application Controller controls the running of applications by: processing input data and instructions/selections from users or other systems; initiating the display or output of information via the Application Viewer, commanding the DGI to initiate application state transitions and controlling the import and export of GCD application descriptions. The Application Controller is the *controller* component of the Dynamic Model View Controller.

The Application Viewer is the *view* component of the Dynamic Model View Controller. It receives display/output commands from the Application Controller to render GCD application data on a display media. The rendered application data provides a means of interacting with the applications via selection and input controls, and a way to view and update their content.

The third element of the Dynamic Model View Controller, is the integrated GCD and DGI, which functions as the *model* component.

In addition to providing application control and view capabilities, the Application Controller Viewer provides a WYSIWYG (What You See Is What You Get) editor for editing application data display windows. Changes made to an application's windows take affect immediately. They can affect all executions of an application, a single window within an application or a particular instance of running an application. An example of an application window displayed by the editor is shown in Figure 3.

2.4 Application Editor System (AES)

The Application Editor System is a fully-featured editing environment for defining and updating EGCD applications. It renders graphical representations of EGCD applications that directly correlate to the actual GCD application structures shown in Figure 1. The graphical representations include symbols and structures that developers manipulate with editing commands entered via a keyboard, computer mouse or other input device. Figure 4 shows an implementation of the PointDragon Visual Programming Language Editor. The symbols in Figure 4 follow the syntax of PointDragon language where the process nodes of the GCD are called Classes and Methods and data definition nodes are called Data Action Maps, Tables, Data Groups and Fields. A more detailed description of PointDragon follows in Section III.

The Application Editor System is implemented by the Model View Controller structure shown in Figure 5. It interfaces with the GCD-DGI via the EGCD-API. The EGCD-API provides a model of an EGCD application that is notation editor oriented. Other notation editors can be interfaced with the GCD-DGI through the EGCD-API by implementing a notation-specific Notation Editor Graph for the editor. The Notation Editor Graph defines the specific syntax of a programming language. In the PointDragon implementation the language is visual.

2.5 Executable Graph Component Database-Application Program Interface (EGCD-API)

The EGCD-API is a standard set of functions and associated parameters implemented by an EGCD Translator. It is composed of

GCD and DGI components that enable non EGCD applications, such as, the Application Editor System, to interact with an EGCD application. It includes functions for starting, stopping, defining and modifying EGCD applications. It also includes functions for inputting data to EGCD applications and reporting their outputs.

2.6 GCD and DGI Descriptions and Interactions

The primary elements of the EGCD are the Graph Component Database and the Dynamic Graph Interpreter.

The GCD is safely mutable while it is being interpreted. It is comprised of data nodes, process nodes, application state nodes, specialized nodes, relational edges, and transformational edges. Data nodes define and store data and include a variable set of properties that indicate the types and data values that may be stored within them. A data node roughly corresponds to the tables and columns of a relational database or to the persistent objects and attributes of an object oriented database.

Process nodes define and store the dynamic processes, temporal actions, ordered or unordered and serializable or parallelizable sequences of actions or steps of a process or processes, and include variable sets of properties that specialize the type of actions or steps the process nodes define. A process node roughly corresponds to a line of code within a high level computer program, an operation or step within a workflow system, or a screen or window within an interactive or transactional system.

Application state nodes store the complete state of an instance of a running application. They hold the application state at each process node (step). They are comprised of data nodes, process nodes and other specialized GCD nodes. An application state node roughly corresponds to the transaction state in a transactional system or the working storage in a traditional program during execution.

Specialized nodes are nodes such as user nodes and audit nodes, which control access, provide security, audit trails, and the backup and recovery functions required for large-scale, complex applications.

Relational edges define a structural relationship between nodes in the GCD, indicating, for example, whether a hierarchical (parent-to-child) or sibling (peer-to-peer) relationship exists between two nodes. They include properties that define the particular structural relationship of an edge.

Transformational edges define a state transforming relationship between two nodes. When acted on, they cause changes to the property values held by two nodes or to nodes structurally related to them via other relational edges. Transformational edges include properties that define the particular transformation to be performed. A transformational edge roughly corresponds to an instruction like MOVE, COMPUTE, CONDITIONAL BRANCH, etc. of a very high level computer language. For example, MOVE can move a complex data structure according to certain rules, and COMPUTE can compute a result with a complex formula using multiple data sources.

A data node may be either a definition or instance data node. A definition data node defines a set of instance data nodes that have common properties. These properties control how the data values of a data instance node, which are a subset of the node's properties, are processed and displayed. For example, display and edit are two properties that indicate how a data value property should be displayed or can be edited via a web browser. Display coordinate properties indicate the location on a web page where values should be displayed; other properties indicate font, color, etc. Properties may also include validation criteria, e.g. the use of regular expressions to validate input. Properties can be process node specific, i.e. each definition node has a default set of proper-

ties, but for any one process node, a property can be redefined specifically for that process node. Any instance node that exists in the application state of an active process node would use its node properties for that process node. If no process node specific properties exist for an instance node, the default properties of its definition node are used. Moreover, properties can be application state node specific, i.e. a property can be defined for a particular application state. This means that the properties of a data definition node may be either process specific or application state specific, and a data definition node may itself be process or application state specific. A data definition node specified in this way will only be applied to a given process or application state or to combinations of process and/or application states, and ignored at all other process steps or application states. Allowing properties of data nodes to be process or application state specific gives EGCD applications tremendous flexibility. This concept enables users to specialize their applications for a specific set of data that they are processing through the application without affecting other users with differing requirements for their specific data sets.

Instance data nodes are specific instantiations of data definition nodes. This is similar to the instantiation of objects in the object oriented paradigm. Indeed, the GCD can be defined as an object oriented repository where the objects are nodes within a directed graph, and new specialized process node and application state node objects have been defined in addition to the other specialized functionality described herein.

Data node properties may also be set at the data instance level. Specific data types, for example, may have their own set of properties. For example, the data type 'file' has file name and path properties, which are instance specific. All data types have at least one value property that is instance specific. Finally, data definition nodes can hold default values for instance specific properties.

Process nodes control the update and use of data instance nodes throughout an application and the creation, modification, and display of data instance nodes via the interactive display media used by an application. Operation nodes, which are one type of process node, represent discrete steps in the data and process flows through the application, including changes to the display media of an application.

The DGI is an interpreter for running computer applications defined by a GCD on a computer system. It can modify GCD's and run GCD's via the Application Controller Viewer and a standard web browser. The interpreter enables a user to safely modify any of the nodes and edges that define an application in the GCD while the application is running. Figures 6 and 7 provide a high level description of how the DGI runs and displays a GCD based application. In Figure 6, the DGI is accessing the GCD. It takes input from the user via the web browser and evaluates the current application state node X, which contains various data instance nodes as well as active and inactive process nodes (see Figure 7 for an expanded application state node). The DGI then accesses the corresponding data definition and process nodes within the GCD to determine how it will process and change the application state node X, according to input from the user as well as the structure of related nodes in the GCD. At the same time, changes to both the application state node X and data definition and process nodes may be affected. Application state nodes X' and X'' in Figure 6 represent changes to application state node X that are made by the DGI as a result of its evaluation.

Figure 7 demonstrates how a user interface is created by the DGI utilizing definitions in the GCD. The DGI examines the data instance data nodes within the application state node, the corresponding data definition nodes in the GCD, as well the process nodes (Operation A in Figure 7), and the corresponding display

properties for the data definition nodes (Figure 7, far right) to create the user interface seen on the web browser. In the case of data instance B, the DGI uses the display properties for application state node X, which override existing display properties for Operation A (the current active process node). Note the lack of arrow in Figure 7 from DGI to “Data B Display Properties for Operation A”. Data instance A does not have specific display properties for application state node X, therefore the display properties for the currently active operation A are used.

3. PointDragon

3.1 Overview

The PointDragon platform is a commercial implementation of an Executable Graph Component Database system. The PointDragon platform is composed of the five elements of an EGCD system described above, the GCD, DGI, ACV, AES and EGCD-API. The AES described above is a key element of the PointDragon IDE contained within the platform. The specific syntax of the object-oriented Visual Programming Language used within the IDE is fully defined by the AES and EGCD-API components. The GCD and DGI form the core of the run-time platform contained within PointDragon, which also includes all the functionality traditionally found in database systems, as well as other functionality critical to running and maintaining applications such as security, access control and auditing.

The PointDragon platform was designed to achieve the following goals:

- **General purpose:** to be as general purpose as tools like Java, Oracle and AJAX; to handle both very simple and very complex applications in any domain; and to be scalable so that it is suitable for both low performance single user applications and large scale, enterprise applications with multi-million row databases and thousands of users.
- **Fusion of Program, Database, and User Interface:** to seamlessly fuse all aspects of application development including program logic, database definition, updating and reporting and user interface creation. The core syntax of the language provides a common approach to both program and database definition, as well as, built in syntax to simplify user interface definition. This concept of fusion applies to other aspects of PointDragon application development and run-time management, e.g. fusing security and administration into the core platform, and the core syntax of the PVPL.
- **Visual:** to be 100% visual – everything is defined by point, click, drag and drop actions. No scripting language is used. Text entry is needed only when labeling programming elements, such as classes, methods, or data elements.
- **Web-based:** to provide a completely web-based visual programming language.
- **Object-oriented:** to exploit object oriented principals, consistent with the three goals immediately above (fusion, visual and web-based).

3.2 PointDragon Visual Programming Language (PVPL) Elements

As described earlier, the Executable Graph Component Database system on which PointDragon is based has three kinds of components: nodes, edges and properties. Nodes can be either composite

or primitive data definitions or instances or function definitions. Edges define relationships or state changes between nodes. Properties are attributes of either nodes or edges that define their behavior.

The basic programming elements of the PointDragon Visual Programming Language (see Figure 4 above), are specializations of the EGCD components described above. Where possible, programming elements were given names that corresponded to elements in other programming languages with similar semantic meaning, however in no instance is the correspondence exact:

- **Classes:** represent complete process elements within an application containing both data and program logic. The PointDragon editor displays all the elements of a class on a single class canvas. An instance of a class is created by a start (constructor) method. Instances of a class are called objects and are automatically persistent. Classes may contain any of the other PointDragon programming elements described below. A class may inherit methods from other classes (multiple inheritance is allowed). The PointDragon editor displays all the Classes of an application on a single Application canvas.
- **Methods:** are program logic compositions within a class. They are composed of a set of Data Action Maps and a Method node. At least one Data Action Map within the set must be linked to the Method node with an Action. All other members of the set must be linked to either the Method node or at least one other member of the set via an Action. A Method may inherit from any other Method in any Class. When a Method inherits from another Method it inherits all the Data Action Maps associated with that method. It may then add additional Data Action Maps of its own to the inherited method. Methods may also invoke other Methods with parameters and receive results. There are two specialized types of Methods: Start Methods and Window Methods. Start Methods are constructors as defined above. Window Methods are Methods that display data elements on a browser. Methods have two execution phases: Initialize and Main. The Initialize phase of a Window Method occurs prior to display. The Main execution phase of a Method is triggered by user input from the browser.
- **Data Action Maps:** are data structures within a class that define data elements and their Actions. Data Action Maps are the smallest encapsulated data/program unit within a class. All Actions within a Data Action Map are executed together (though a Data Action Map can include Actions executed in either or both the Initialize or Main phases of a Method). Data Action Maps within Window Methods are automatically persistent.
- **Tables:** are tabular data sets within the Database or within a Data Action Map or Data Group. Tables within the Database are automatically persistent and share strong similarities with tables in traditional relational databases. Data Action Maps or Tables in a Data Action Map may be joined to Tables in the Database to select a single row or set of rows from the Database Table. The row or set of rows may be manipulated as an independent data structure within a Method.
- **Data Group:** are structured sets of data elements, which may contain Fields, Tables or other Data Groups. Data Groups may be components of Data Action Maps, Tables or other Data Groups.

- **Fields/Columns:** Fields are elements of Data Action Maps and Data Groups that hold predefined data objects. Columns are Fields within a Table. The predefined data objects contained within fields range from simple objects that hold text or integer values to complex objects like email, web link or web service that perform a series of predefined actions depending on the data elements to which they are linked via actions.
- **Actions:** define a dynamic relationship between two components (Methods, Data Actions Maps, Tables, Data Groups and Fields/Columns). The components being linked determine the type of Action that links them. For example a Display Action links two Methods, but a Move To Action links a data element (Data Action Map, Table, Data Group, or Fields/Columns) to another data element. The dynamic relationship defined by an Action is determined by both the type of component and the type of data objects that the component might hold (if it's a data element).
- **Action Conditions:** define conditions for executing Actions. An Action Condition may be placed on most Actions. It holds a logical condition test between two data elements. In addition, the Condition may be used to execute a Method while the Action is being performed.
- **Menus:** are specialized objects that hold Classes. The Window Methods of Classes that are contained by Menu objects can then become drop down menu items within an Application.

3.3 PointDragon Action Types

Action types fall into three categories: Method to Method Actions, Data to Data Actions and Method to Data Actions. The descriptions provided below are brief summaries of the functionality associated with each Action type.

3.3.3 Method to Method Actions

- **Display:** A Display action moves an object, which represents an application state, from one method to another method and immediately display the user interface of the second method. A Display action can be made conditional by dropping an Action Condition on it.
- **Queue:** A Queue action moves an object from one method to the queue of another method and displays the workflow object. A method can be the source of an arbitrary number of Queue actions. Queue actions can be made conditional by dropping an Action Condition on them.
- **Execute:** An Execute action completely executes a method before any other method is queued or displayed. This can be interpreted as the system both initializing and submitting from the executed method. An Execute action can be made conditional by dropping an Action Condition on it.
- **Inherit:** An Inherit action allows one method to appear and behave exactly the same as another method. All of the inherited method's data objects and associated actions, excluding method to method actions, are executed in the inheriting method.
- **Inherit Display:** An Inherit Display action applies the user interface properties of one method to the user interface proper-

ties of another method. User interface properties include display, edit, and WYSIWYG settings.

3.3.3 Data to Data Actions:

- **Move To:** A Move To action transfers data between Data Action Maps, Tables, and Fields/Columns. The effect of a Move To action on data varies according to the types of programming elements that are linked as well as the data types of the programming elements, in the case of Fields/Columns.
- **Get From:** Similar in action to a Move To action except the transfer is in the opposite direction and the order of execution within the Method may be different.
- **Join:** A Join action can be made from a Field/Column in a Data Action Map/Table to a Field/Column in its associated Database Table (Row). This allows users to either store new data into a Database Table or edit existing data in a Database Table. Typically, Join actions are connected between uniquely identifying Fields/Columns – such as a sequence number or instance number. It ensures that a single instance is retrieved for the object.
- **Retrieve:** A Retrieve action is used to extract one or more data rows from a Database Table. Retrieve actions can also be used to validate the existence or uniqueness of a value against a database table column.
- **Math Operation:** A Math Operation action performs simple numeric computations. The Math Operation action can be linked from one numeric Field/Column to another or to itself. The result of the operation is transferred to the Field/Column that the action points to.
- **Execute Data Action Map:** An Execute Data Action Map action aids in the control of the flow of execution. It provides the ability to execute Data Action Maps upon submit, loop the execution of a Data Action Map, and iterate through Tables. An Execute Data Action Map allows the creation and execution of one Data Action Map to be invoked by another Data Action Map. When the action executes, everything within that linked Data Action Map - including Get From actions and Move To actions – also execute accordingly. These actions can also be used to construct a loop that executes the same Data Action Map over a specified number of iterations. Finally, the Execute Data Action Map can be used to execute a data action map for each row of a Table.
- **Split:** A Split action is used to generate one or more new objects at a subsequent Queue. When linked from a Data Group to a Table, each row of the table is transferred to a single data group within a new object. This is the equivalent of producing multiple Object IDs from a single Object ID's table.
- **Merge:** A Merge action is used between a Table and a Data Action Map. At a Merge method, the data action maps of multiple objects can be combined to form a single new Table and a new object.
- **Foreign Key:** Database Tables can be made relational using Foreign Key links. This allows for related rows to be cross referenced and carried together in an object.

- **Compute With:** A Compute With action links a field/column to the computed field when a formula is created on the computed data field/column. These actions are automatically created for each variable included in the formula.

3.3.3 Method to Data Actions:

- **Create Data:** A Create Data action connects a Method to a Data Action Map by default. Data Action Maps are created prior to display and execution. A Data Action Map always executes at the Method from which it is created, though it can be displayed on subsequent user interfaces as well.
- **Execute From Data:** An Execute from Data action is used to connect a Field/Column of data type “web link” to a Method. It transforms the Field/Column into a clickable web link on the Application Interface. Clicking on the web link will pass the user to the method to which it is connected.

3.3.3 PointDragon Visual Programming Language and Object Oriented Programming

The PointDragon Visual Programming incorporates the standard characteristics of an object oriented programming language (OOPL), including encapsulation, inheritance, polymorphism, composition, classes, object instances, attributes, and methods. However, there are also many differences between PVPL and classic object oriented programming languages like Java. In defining the language an effort was made to exploit both the visual nature of the language as well as the underlying EGCD architecture. In particular the EGCD architecture allows tight and comprehensive integration of the disparate parts of a software system in an atypical manner.

- **Computing Platform Integration:** Like some existing OOPLs, PVPL is tightly integrated within an IDE. However, the PVPL is also tightly integrated within the PointDragon Runtime Platform and Database that is used to manage the production execution and maintenance of PVPL based systems in addition to the development of such systems.
- **Visual Programming Language:** PVPL, unlike most OOPLs, is a Visual Programming Language. No scripting is required to use any part of the PVPL functionality. All language features are completely accessible within the PointDragon platform.
- **Logical Connections between Methods:** All class Methods are specialized nodes within the Executable Graph Component Database. As a result, the PVPL naturally defines conditional logical relationships and complex process flows between methods graphically via edges. This kind of syntax is only manageable with a VPL and is not generally part of an OOPL.
- **Structured Attributes and Structured Attribute Arrays:** PVPL provides the capability to create attribute structures and arrays of attribute structures. Structures are a common feature of many languages.
- **Large Number of Primitive Data Types:** The system includes many predefined primitive data types. In addition to common primitives like integer and string, examples of PVLP data types include: file, XML and several internet aware data types such as email, URL or web service. PVPL data types have a rich set of associated functionality, which is partly controlled

through the use of properties as described below. Many of these data types could be considered full fledged classes that might be defined within a SDK in an OOPL like Java. PVLP makes heavy use of polymorphism, with the behavior of different edge actions dependent on data type. The primitive data structures with different data types within PVPL may be considered predefined classes – especially for purposes of formalizing the language syntax.

- **Attribute and Data Type Specific Properties:** PVPL includes a large number of predefined properties that can be used to control the behavior of attributes, including the behavior of primitive attributes specific to a particular data type. In addition to the polymorphism described above these properties help define application behavior in many common situations. If one considers primitive data structures within PVPL as classes, properties may be considered predefined attributes for these classes which control the behavior of their corresponding object instances upon assignment. Considering move, get and retrieve actions as assignment statements. PVPL uses overloading that is not only dependent on data type and attribute structure, including array structure, but also on predefined attributes (properties).
- **Method Specific Display Properties for Attributes:** All attributes contain display properties that can be used to precisely define the way an attribute will appear on the user interface. So, for example, one can define whether the value contained in an attribute is editable and its position, size and font on a window. These display properties are defined with the WYSIWYG editor which is part of the PointDragon Platform and are part of the PVPL. In addition, display properties can be defined specifically for any method. With the view that primitives are classes and properties are predefined attributes, PVPL allows for method-specific, predefined attributes that control the behavior of assignment statements within that method. Display properties may also be considered within this framework. The browser user interface, for example, may be considered as a set of predefined classes and specific instances of these classes as specific screens, ultimately updated with various objects via assignment statements.
- **Object and Attribute Level Tables:** All object instances created within a PVPL application are automatically and transparently persistence. However, this can be overridden for transient working storage elements. Although object oriented databases exist that work with OOPLs to save instances of persistent classes, they normally don’t automatically and transparently persist all object instances. Such automatic persistence facilitates and simplifies the construction of a wide variety of complex web based applications.
- **Persistent Public Attribute Tables:** Databases of persistent public attribute tables can be created within the PVPL. These attribute tables can be updated and queried directly with the PVPL – from any method. These attribute tables allow for the simple saving and retrieving of data in a fashion similar to the relational database model. They are easily mapped to external relational tables. Access to these attribute tables can be restricted by user, class or method per table or attribute within a table. These persistent public attribute tables, together with the automatic object and attribute persistence described above, provide the PVPL with the full complement of functionality seen in relational models within an OOPL framework. How-

ever, unlike many other OOPs, no object – relational mapping is required to interface with a relational model.

- **Testing Environment Integration:** The PVPL is completely integrated within an automatic testing environment. A program change made in the PVPL can automatically and transparently be tested within this environment with no additional work. Predefined attribute properties can be used to control test values.

4. Conclusion

The PointDragon platform and its underlying Executable Graph Component Database have been used to build and run a number of large, complex systems at biotech and pharmaceutical companies. Anecdotaly, the system seems to provide very large development and maintenance productivity improvements over traditional software development systems. However no systematic, controlled experimentation has been done yet, and in fact, the PointDragon implementation has been evolving too rapidly for such controlled experimentation to take place. It is presented here as a platform with various novel attributes, which the reader is invited to try [5].

The PointDragon implementation presented here is just one possible example of an EGCD system. We believe the EGCD can facilitate the implementation of a range of very novel and powerful programming languages, development environments, databases and run-time platforms.

Acknowledgments

We thank Vladimir Gusev and Hongping Liang for their contributions to the implementation of an Executable Graph Component Database system.

References

- [1] S.A.Gold, D.M.Baker, V.Gusev, and H.Liang. Object Process Graph System. United States Patent 7,316,001, January 2008. GraphLogic Inc.
- [2] S.A.Gold, D.M.Baker, V.Gusev, and H.Liang. Object Process Graph Application Controller-Viewer. United States Patent Application 2006/0015857, January 2006. GraphLogic Inc.
- [3] S.A.Gold, D.M.Baker. Object Process Graph Relational Database Interface. United States Patent Application 2006/0004851, January 2006. GraphLogic Inc.
- [4] S.A.Gold, D.M.Baker, V.Gusev, and H.Liang. Object Process Graph Application Development System. United States Patent Application 2006/0059461, March 2006. GraphLogic Inc.
- [5] PointDragon website and PointDragon IDE and run-time platform. GraphLogic Inc. March 2008. www.pointdragon.com.
- [6] Rapid RNAi Informatics Development with PointDragon™, A. Birch, Merck & Co Inc., April, 2008. <http://www.bio-itworldexpo.com/track2.asp>.
- [7] M.M. Burnett, A. Goldberg, and T. Lewis, 1995. Visual Object-Oriented Programming. Manning Publications, Greenwich, CT.
- [8] N.H. Balkir, G. Ozsoyoglu, and Z.M. Ozsoyoglu, 2002. A Graphical Query Language: VISUAL and Its Query Processing. *IEEE Transactions on Knowledge and Data Engineering*. 14:5: 955-978.
- [9] G. Costagliola, A. Delucia, S. Orefice, and G. Polese, 2002. A classification Framework to Support the Design of Visual languages. *Journal of Visual Languages and Computing*. 13: 573-600.
- [10] K. Zhang, D. Zhang, and J. Cao, 2001. Design, Construction, and Application of a Generic Visual Language Generation Environment. *IEEE Transactions on Software Engineering*. 27:4: 289-307.
- [11] ObjectDB website, ObjectDB, March, 2008. www.objectdb.com.

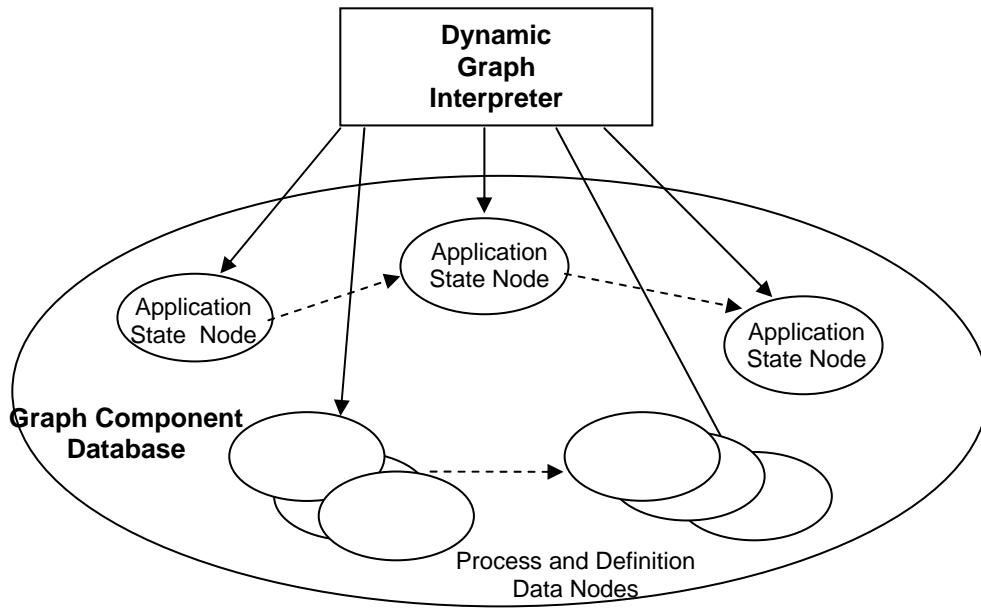
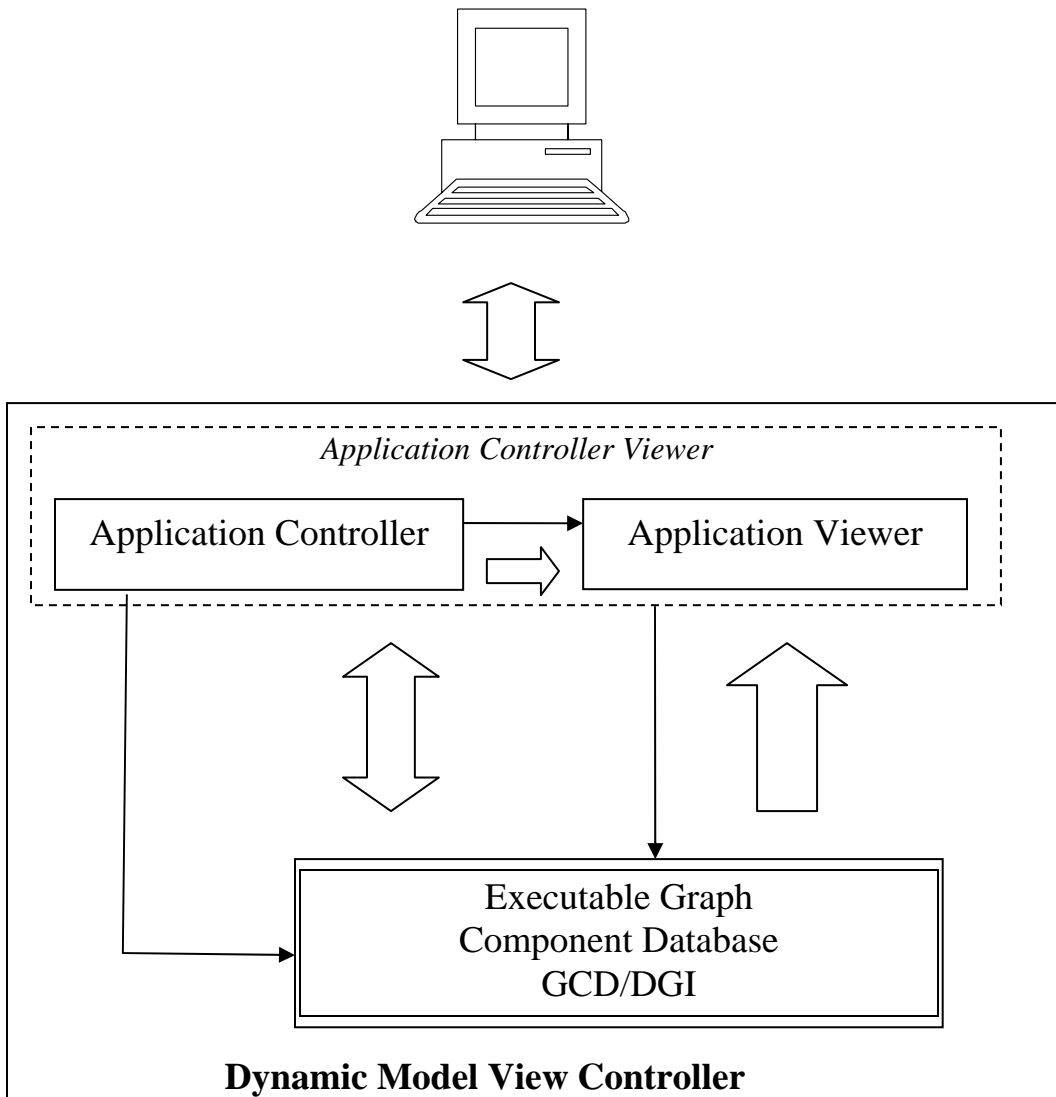


Figure 1. Dynamic Graph Interpreter accessing a Graph Component Database to execute an application



Jakarta Struts WEB Application

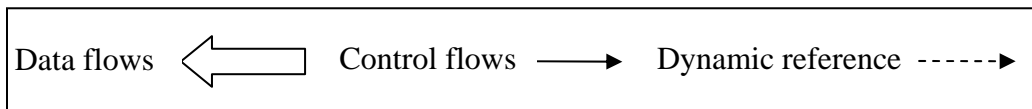


Figure 2. Dynamic Model View Controller (DMVC)

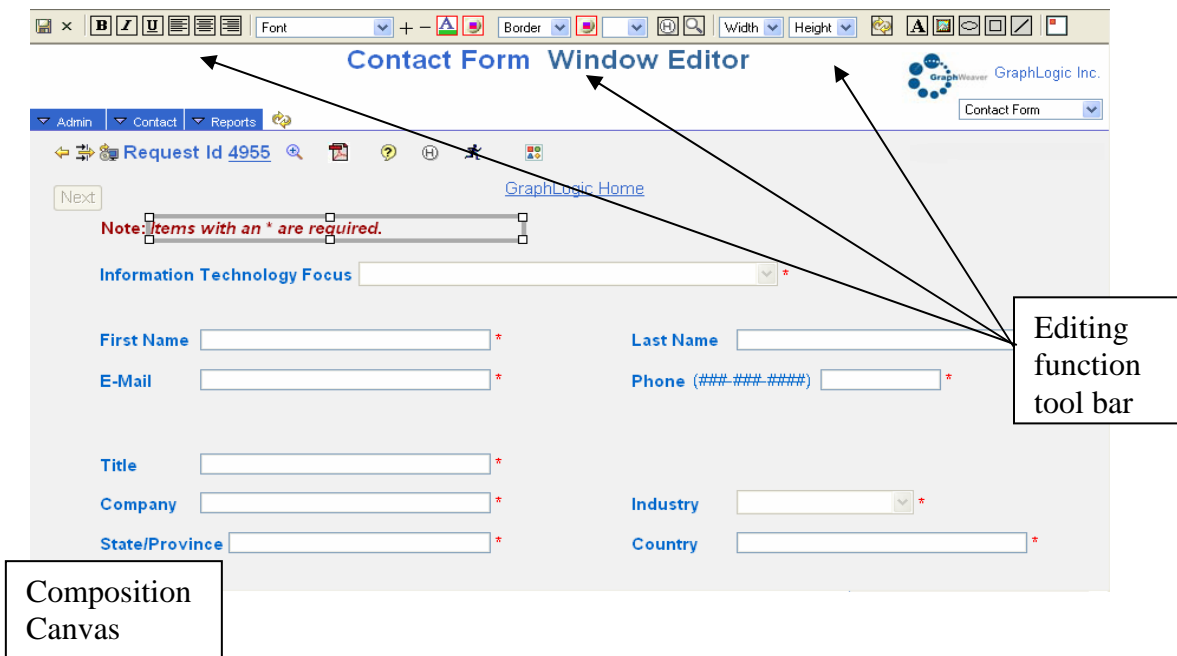


Figure 3. Application Window Editor

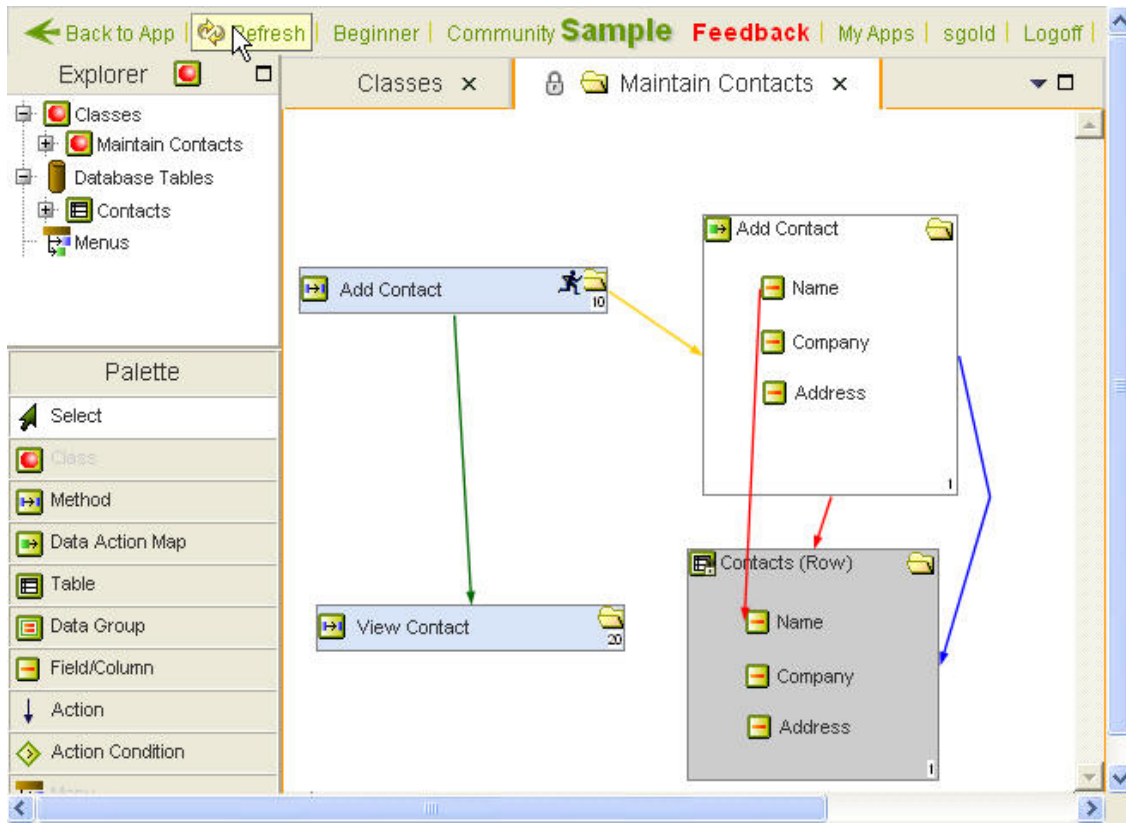


Figure 4. EGCD Application Editor System (The PointDragon Implementation of an EGCD system)

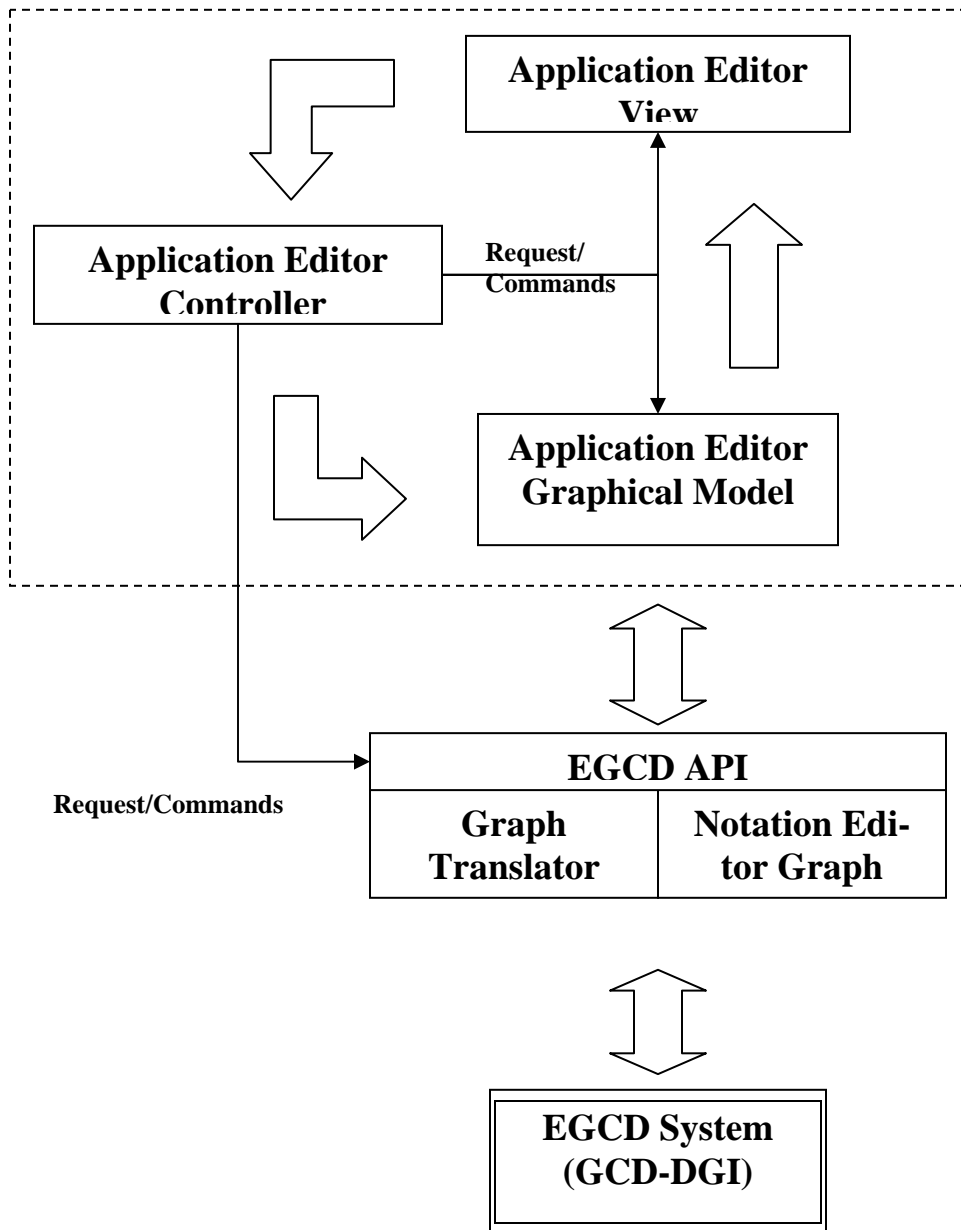


Figure 5. Application Editor System and Executable Graph Component Database Application Program Interface (EGCD-API)

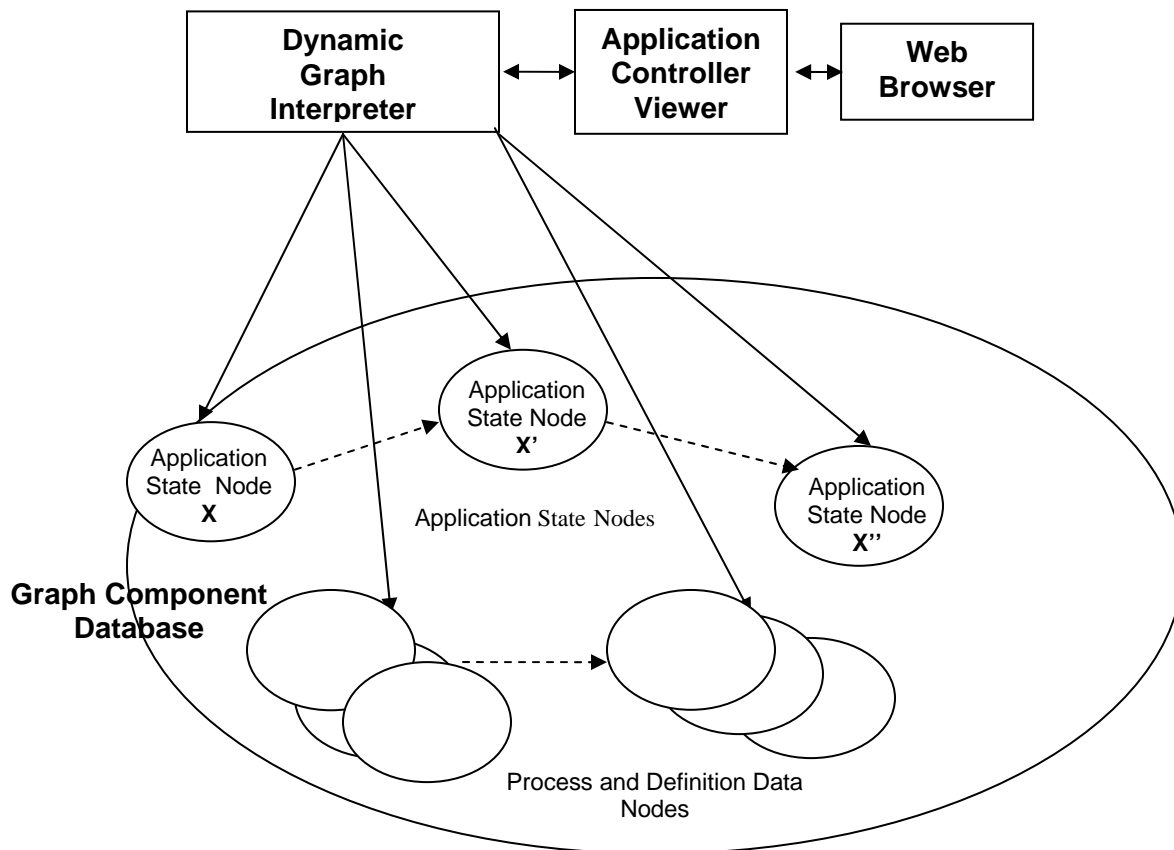


Figure 6. Dynamic Graph Interpreter accessing a Graph Component Database to execute an application through a web browser.

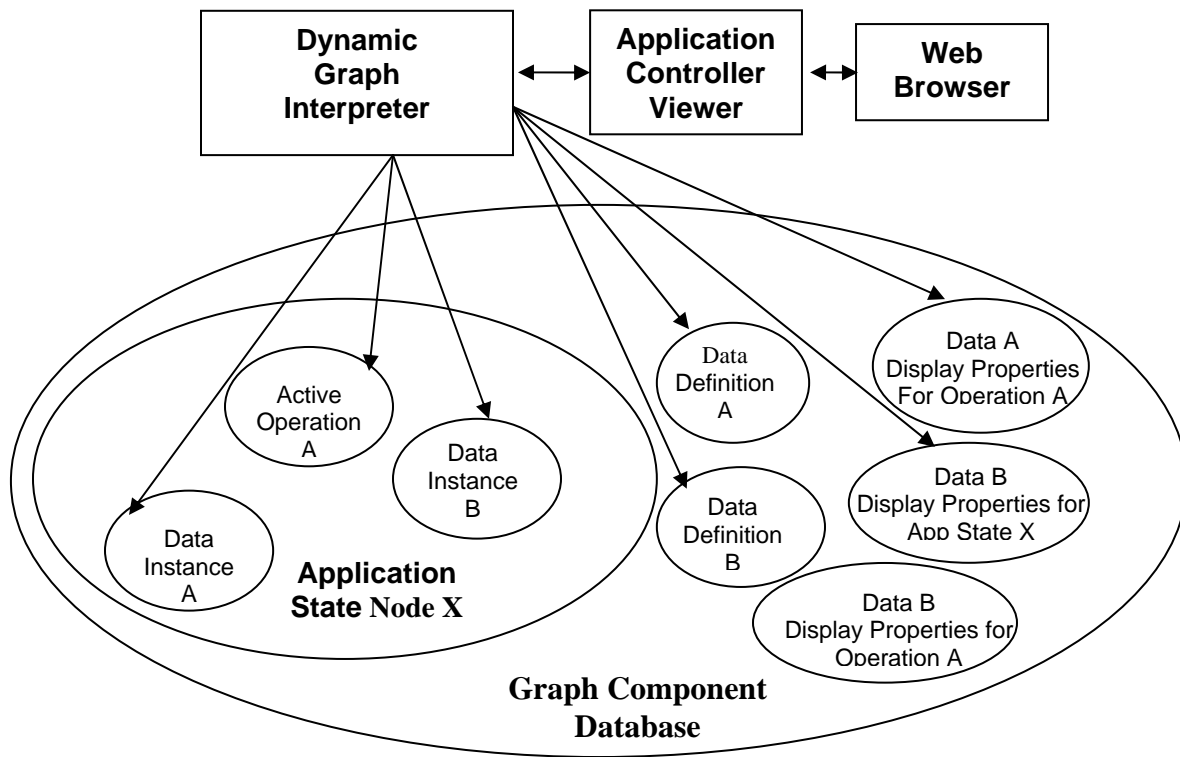


Figure 7. Dynamic Graph Interpreter accessing display properties of data nodes within the Graph Component Database to construct a User Interface on a web browser.