Software Framework Concepts for Power Distribution System Analysis

Fangxing Li, *Member, IEEE,* and Robert P. Broadwater, *Member, IEEE*

***Abstract*—Software design reuse has been discussed in the past at the component level through Design Patterns. A software framework is an approach to achieve software reusability for an entire domain. This paper presents architectural design concepts of a framework for power distribution system analysis. The commonalities of distribution system analysis, including components, topologies, and algorithms are considered. A layered architecture with strict top-down dependency is proposed to decrease software couplings. Interfaces are used to hide the internal structure of each layer. The Composite and Iterator design patterns are used in the framework design. This paper also presents practical examples of developing customized applications that reuse and extend the framework.**

***Index Terms*—Design patterns, power distribution systems, software frameworks, software reusability.**

I. Introduction

In the past several decades many software packages have been developed for electric power systems. Object-oriented programming (OOP) and object-oriented design (OOD) have improved software development for power system analysis [1]–[6]. These previous works presented approaches of software design and development for power system applications. However, there were many repeated works in software development, particularly in the design area.

Design patterns provide for reuse of software design [1], [7]–[9]. Previous work [1] discussed applications of design patterns for modeling components in power systems. It focused on many small, individual design modules since patterns usually target specific design problems. Also, it addressed components only, and did not address topology or system analysis algorithms. This paper addresses reusability of OOD at the domain level. Here the domain to be addressed is power distribution analysis, including component models, topological traverses, and analysis algorithms. The reusability is achieved with a software framework that summarizes the commonalities in distribution system analysis. The features of the framework are as follows.

- The framework serves as a platform for software engineers who work in power distribution applications.
- The architecture is layer-driven and top-down dependent. That is, higher layers depend on lower layers, while lower layers do not depend on higher layers. Therefore, the structure is levelizable [5], [10].
- Standard design patterns are applied for the internal design of layers to achieve maintainability.
- The dependency among layers is driven by interfaces to further reduce software coupling and to achieve encapsulation for the layer associated with the interface.
- Interfaces are defined in a language-neutral standard to make the framework essentially accessible to applications developed in heterogeneous languages.

Manuscript received May 29, 2003.

F. Li is with ABB, Inc., Raleigh, NC 27606 USA (e-mail: fangxing.li@us.abb.com). He was with the Electrical and Computer Engineering Department, Virginia Tech, Blacksburg, VA 24061 USA.

R. P. Broadwater is with the Electrical and Computer Engineering Department, Virginia Tech, Blacksburg, VA 24061 USA (e-mail: dew@vt.edu).

Digital Object Identifier 10.1109/TPWRS.2003.821437

II. Fundamental Concepts

A. Concept of Frameworks

In the scope of object-oriented design, a framework is a reusable design that is decomposed into a set of cooperating classes, which can be specialized to produce custom applications [7], [8]. A framework is object-oriented by definition. A framework usually dictates a fundamental architectural design for applications that are incorporated into the framework. A framework predefines many mundane design parameters and actions so that an application developer can concentrate on the specifics, which make the application unique from other applications in the problem domain. This implies that a framework can be viewed as a semi-completed application. This also implies that the users of a framework are application developers in the domain of the framework.

Frameworks have been used to achieve software reuse in a number of domains. Some examples are listed as follows [9].

- Common Object Request Broker Architecture (CORBA) for distributed computing
- FIONA and MultiTEL for communication systems
- Extensible Computational Chemistry Environment (ECCE) for computational chemistry
- G++ for Computer Integrated Manufacturing (CIM)

Frameworks may be viewed as evolved from class libraries, which were presented during the early age of object-oriented technology. Also, class libraries may be viewed as evolved from function libraries, which were presented before the object-oriented age. Fig. 1 illustrates the evolution process of these three techniques. It is worthwhile to mention that a legacy class library could certainly be refined with the latest object-oriented technologies and demonstrate many framework-like features.

B. Scope of the Proposed Framework

This paper presents the high-level design of a software framework that abstracts the common attributes and behaviors in power distribution system analysis. The users of the

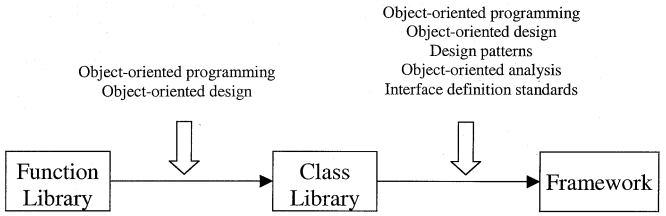

Fig. 1. The evolution process of functional library, class library, and framework.

framework are independent application developers in power distribution analysis. Since the framework is designed to obtain high reusability and extensibility, the application developers do not need to create their application from scratch. Instead, they may reuse and extend the framework based on their own needs.

Based on the above objective, the framework is designed to model only common distribution system algorithms, such as load flows, short circuits, and reliability assessment. More specific algorithms, such as optimal reconfiguration, risk analysis, and others, should be addressed by the framework users or application developers. To model the generic distribution analysis, component and system topology must be modeled as well. Also, with the consideration of the increasing interest in applying the Internet and distributed computing to carry out transmission and distribution analysis [11]–[13], this framework provides an interface for distributed computing with the concurrent server-client mode. In summary, the scope of this framework includes the following.

- To present a multiphase model for distribution components such as lines, cables, transformers, and others.
- To model topological structure and traversal methods of distribution systems.
- To model fundamental algorithms for distribution analysis, such as load flows, short circuits, and reliability analysis.
- To model distributed computing to allow applications running at different machines in client-server mode.

C. Distribution System Models

A majority of US distribution systems are radial or weakly meshed. A weakly meshed system can be viewed as a set of radial systems that are interconnected with a few components, usually switches. Hence, distribution systems can be essentially modeled as radial trees.

System components are presented with a two-node model as shown in Fig. 2. In other words, each component contains a single upstream port, or port 1, and a single downstream port, or port 2. The upstream port is closer to the power sources, while the downstream port is closer to ending customer loads. Voltage, current, and load flow at each port are considered as internal attributes of each component. With this approach, there is no need to use a node-edge-based topological representation or an incident matrix approach that models nodes and edges separately. Hence, this approach reduces topological complexity and simplifies implementation. It should be noted that the downstream port of a parent component is the same port of the upstream port of any of its child components. This implication makes it possible to build system connectivity (or topology) and eliminate the need of an incident matrix.

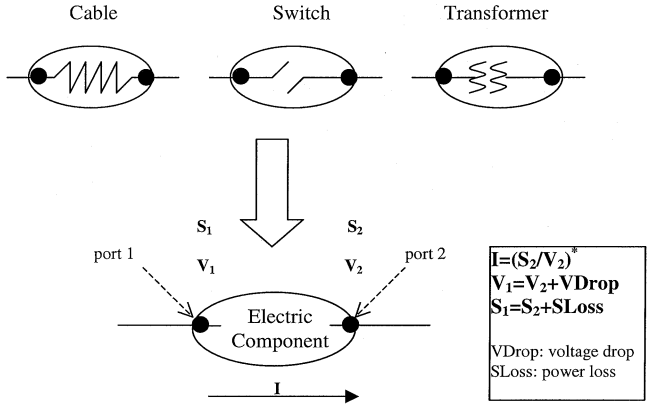

Fig. 2. Two-port component model.

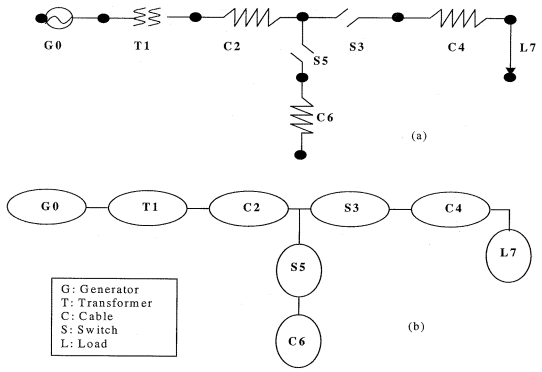

Fig. 3. Two representations of a power distribution system.

Components like a three-winding transformer may have three nodes connected to a system. In this case, it can be modeled as three two-winding transformers, each of which is a two-node component.

With the above two-node component model, distribution systems can be modeled as radial trees from the topological point of view. Figs. 3(a) and (b) illustrate node-edge-based and component-based representations for a simple system, respectively. In Fig. 3(b) the two internal ports are not shown for simplicity. Also, "lines" or edges in Fig. 3(b) represent interconnectivities and do not correspond to components in the system.

The above component model and topological tree model are commonly used in distribution analysis [3], [14]–[17], such as load flows, reliability assessments, and system reconfigurations. Here the component and topological models exist as part of the framework and are shared by all applications.

III. Architectural Design

This section discusses architectural design of the framework. From the external standpoint, the framework is designed to operate across heterogeneous environments through Object Management Group (OMG)'s Interface Definition Language (IDL) [18]. As a script-oriented, language-neutral standard, IDL can be mapped to many object-oriented languages such as C++ or Java. This gives significant flexibility and interoperability to the framework. As shown in Fig. 4, the framework, implemented

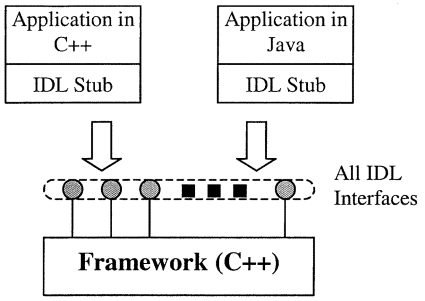

Fig. 4. Using IDL interfaces as external representation of the framework for heterogeneous applications.

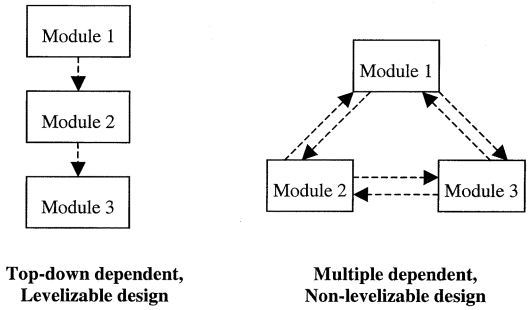

Fig. 5. Dependency and levelizability.

in C++, provides IDL interfaces through which the inside modules are called. This makes it very convenient for the framework users (i.e., the application developers) to develop their own applications in other languages.

From the internal standpoint, the framework is designed with a layered architecture consisting of multiple, top-down dependent layers, as shown in the left part in Fig. 5. Compared with a more highly coupled design as shown in the right part in Fig. 5, the top-down dependency is more levelizable and hence more reusable since less dependency is involved among different modules.

The framework consists of four layers, *Component* layer, *Iterator* layer, *Algorithm* layer, and *DistributedAlg* layer, from bottom to top. Each layer is essentially a package, located at its own level in the overall levelized architecture. A layer may consist of multiple sub-packages, and each sub-package contains a set of co-operating classes. This layered architecture has a strict top-down dependency and each layer depends on its lower layers only. A lower layer in the framework provides more fundamental objects than a higher layer does. This makes a lower layer more reusable than a higher layer.

The dependency between layers is carried out through a set of interfaces, which is the only public part of a layer. The internal structure is hidden from the framework users who have access to the interfaces only. Therefore, encapsulation at the layer level is achieved.

Object-oriented analysis and design (OOAD) of the framework involves the following:

1) Common responsibilities, or behaviors, of all domain objects in a layer are summarized. These behaviors result in the interfaces for the layer.
2) Classes in this layer are identified. These classes are categorized into the inheritance architecture, based on their physical models and their roles in power distribution systems.

A. Component Layer

There are many types of electric devices, or components, in power distribution systems such as cables, transformers, switches, loads, and so on. The responsibilities of this layer are classified into different groups as follows.

- Get/set local connectivity, i.e., directly connected components.
- Get/set three-phase load flow related parameters such as voltage, current, power, and so on.
- Get/set reliability related parameters such as temporary failure rate, sustained failure rate, open circuit rate, mean time to switch (MTTS), mean time to repair (MTTR), and so on.

These different responsibilities are modeled as different interfaces, *IConnectivity*, *ILoadFlow*, and *IReliability*, respectively. Next, IDL is used to sketch these interfaces.

The interface *IConnectivity* is briefly illustrated as follows, where *AbsCmp* is defined as the parent class of all component classes.

```
interface IConnectivity{
  /* getter methods */
  AbsCmp getParent();
  // get parent component
  AbsCmp [] getChildren();
  // get child components
  /* setter methods */
  void setParent([in] AbsCmp e);
  // set parent component
  void setChildren([in]AbsCmp[] cmps);
  //set child components.
}
```

Based on the *IConnectivity* interface, each component knows its parent (upstream) component and child (downstream) components. Hence, it is possible for the *Iterator* layer to identify the system-level topology and traverse the whole system, as to be shown in Section III-B.

The interfaces of *ILoadFlow* and *IReliability* are also illustrated below. Here only the basic getter methods are shown, while the associated setter methods are not shown.

```
interface ILoadFlow{
  /* getter methods */
  Complex [3] getI();
  //current through components
  Complex [3] getV1();
  //voltage at internal port 1
  Complex [3] getVDrop();
  //voltage drop across component
  Complex [3] getV2();
  //voltage at internal port 2
  Complex [3] getS1();
  //load flow at internal port 1
```

```
  Complex [3] getSLoss();
  //power loss in component
  Complex [3] getS2();
  //load flow at internal port 2
  ...
  /*Complex is a predefined data type for
  complex numbers.*/
  /* setter methods are not shown here */
}
interface IReliability{
  /* getter methods */
  double getTempFR();
  //temporary failure rate
  double getSustFR();
  //sustained failure rate
  double getOpenCkt(); //open circuit rate
  double getMTTR(); //Mean time to repair
  double getMTTS(); //Mean time to switch
  ...
  /* setter methods are not shown here */
}
```

Next, all components are placed into a class inheritance hierarchy. *AbsCmp* is an abstract root class for the *Component* layer. This abstract class implements the *IConnectivity*, *ILoadFlow*, and *IReliability* interfaces. From the viewpoint of implementation, these interfaces defined in IDL must be mapped to the language in which the framework is programmed. This is illustrated in Fig. 6, where gray circles represent the interfaces in IDL and white circles represent the interfaces in the framework's language mapped from the IDL interfaces.

Then, components are grouped into major categories like transformers and cables, represented by abstract classes *AbsTxfm* and *AbsCable*. They are further grouped into concrete subclasses as shown in Fig. 6. For example, the *AbsCable* class has subclasses *SinglePhaseCable* and *ThreePhaseCable*.

The Composite design pattern [1], [7] is applied to the *Component* layer as shown in Fig. 6. The Composite pattern treats a "whole" object the same as many individual "part" objects, since both the "whole" and the "part" follow a common parent class. For example, a *Substation* object may contain multiple individual components like *Transformer* and *ThreePhaseCable*, while *Substation*, *Transformer*, and *ThreePhaseCable* all inherit *AbsCmp* class. The Composite design pattern is also applied to *ThreePhaseCable* and *SinglePhaseCable* classes as shown in Fig. 6.

B. Iterator Layer

The *Iterator* layer presents an abstraction of topological management to traverse power distribution systems according to prescribed traversal rules. As discussed in Section II-C, tree-based representation is utilized in the framework design to traverse distribution systems. For a weakly meshed system, a set of radial trees and co-tree components are used to represent system topology. Here a co-tree component usually is a switch connecting two feeders or laterals.

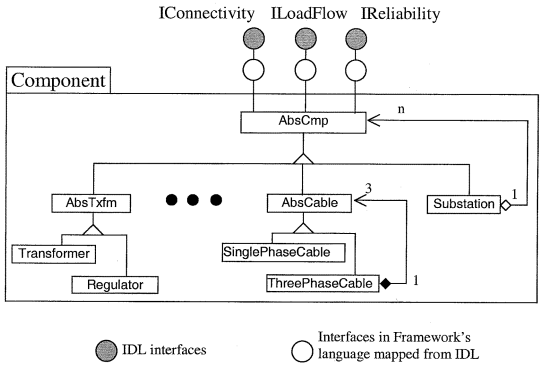

Fig. 6. *Component* layer.

The responsibility of this layer is to traverse all components one-by-one based on specific rules, which could be a forward traverse or a backward traverse. The interface of this layer, *ITraverse*, is as follows.

```
interface ITraverse{
  /* Basic methods related to traverse */
  void setGraph(AbsGraph g);
  AbsCmp firstCmp();
  AbsCmp lastCmp ();
  AbsCmp nextCmp ();
  AbsCmp previousCmp ();
  AbsCmp currentCmp ();
  AbsCmp[] cotreeCmp (); //connecting com-
  ponents
  /* The other setter/getter methods for
  class attributes are not shown here */
}
```

Next, the internal structure of this layer is considered. The Iterator design pattern [7] is employed here to separate the structure representation of a graph and the traversal methods operating on the graph. As shown in Fig. 7, this layer contains two sub-packages, *Graph* and *Traverser*. The *Graph* package contains an abstract parent class *AbsGraph*, which has two subclasses, *Tree* and *Network*, representing radial systems and meshed systems, respectively. The *Traverser* package contains an abstract parent class, *AbsTraverser*, and two direct subclasses, *TreeTraverser* and *NetworkTraverser*. These subclasses may have their own subclasses, *FwdTreeTrav* (for forward traces), *BkwdTreeTrav* (for backward traces), *FwdNetTrav*, and *BkwdNetTrav*. As shown in Fig. 7, each graph may be associated with multiple traversers, while a traverser is associated with only one graph. The traverser classes depend on graph classes.

The Iterator design pattern de-couples the structure representation and the traversal methods of a system. Hence, more traversal methods, such as depth-first forward traverse and breath-first forward traverse [16], can be added without modification of the graph structure and without affecting any existing methods of traversal.

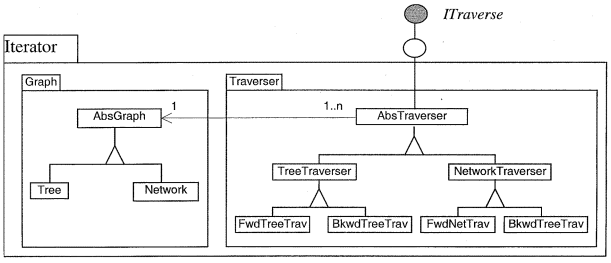

Fig. 7. *Iterator* layer.

C. Algorithm Layer

This layer models the fundamental and well-known analysis algorithms like load flow, short circuit, and reliability assessment. High-level algorithms, like distribution network reconfigurations and optimizations, are not in the scope of the framework. These high-level algorithms are built on top of the fundamental algorithms. They represent what framework users or application developers need to develop.

This layer provides an interface to execute algorithms for a power distribution system, which is represented in a graph and its associated traversers. This interface, *IExecAlg*, is as follows.

```
interface IExecAlg{
  /* Basic methods for starting an algo-
  rithm */
  void setTraverser([in]AbsTraverser
  trav);
  void setGraph([in] AbsGraph g);
  void execAlg ([in] int algID);
  /* The other setter/getter methods for
  class attributes are not shown here. */
}
```

The classes in this layer are grouped into three sub-packages that deal with load flows, short circuits, and reliability assessments, respectively. The basic architecture of this layer is shown in Fig. 8.

D. DistributedAlg Layer

The above three layers are sufficient for developing power system applications in a stand-alone machine. The fourth layer, *DistributedAlg*, provides an abstraction of distributed computations.

The framework should gracefully handle details of distributed computation. The computer network should be transparent to application developers, or framework users. In other words, they should not struggle with programming details of communication protocols. Hence, an interface to invoke remote methods is provided by this layer. This interface, *IRemoteMethodInvoker*, is shown in IDL as follows.

```
interface IRemoteMethodInvoker{
  /* Basic methods for starting dis-
  tributed computing */
  void invokeRemoteAlg ([in] IExecAlg alg,
  [in] long serverID);
```

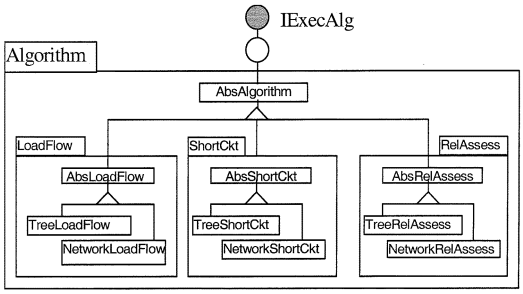

Fig. 8. *Algorithm* layer.

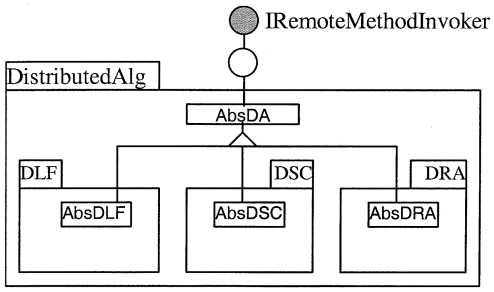

Fig. 9. *DistributedAlg* layer.

```
  void acceptRequest([out] IExecAlg alg,
  [out]long clientID);
  void sendData ([in] AbsTraverser t);
  void sendData ([in] AbsCmp[] cmps, [in]
  long cmpSize);
  void receiveData([out] AbsTraverser t);
  void receiveData([out] AbsCmp[]
  cmps,[out] long cmpSize);
  void execLocalAlg ([in] IExecAlg alg);
  /* Setter/getter methods for attributes
  are not shown here. */
}
```

The class hierarchy is shown in Fig. 9, where *DLF*, *DSC*, and *DRA* are three packages handling distributed load flows, distributed short circuits, and distributed reliability assessments, respectively. *AbsDA* is the abstract parent class for distributed algorithms, and *AbsDLF*, *AbsDSC*, *AbsDRA* are direct subclasses of *AbsDA* in each sub-package. Detailed class diagrams in each package are not shown here for simplicity.

E. System Architecture and Dependency Diagram

Fig. 10 illustrates the system architecture consisting of four layers. As shown by the dashed arrows, this system has a top-down dependency architecture. A dependency relation is carried out through the interfaces associated with a layer, not through the whole layer.

The round rectangle shows the boundary of the framework. The gray circles are the IDL interfaces which make the framework callable to heterogeneous applications. As previously mentioned, the white circles represent the internal interfaces mapped from IDL interfaces. It should be noted that additional design like object instantiation is necessary to make the framework fully callable to heterogeneous applications. Since

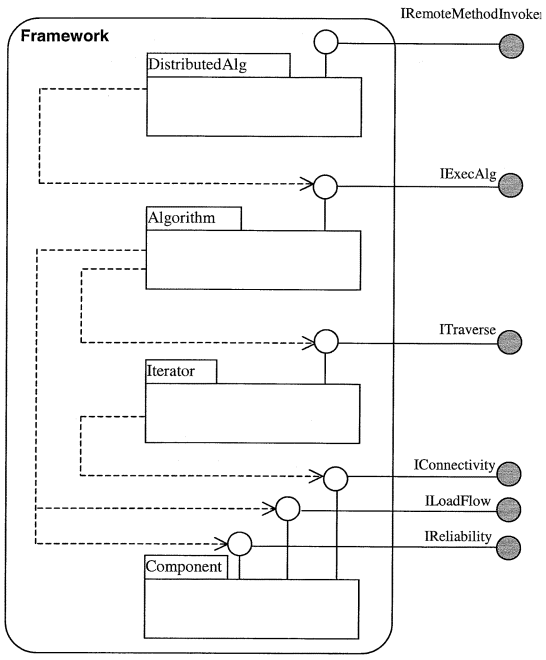

Fig. 10. Framework architecture.

this is not related to the targeted domain—power distribution analysis—it is not shown in this high-level, system design.

IV. Creating Customized Applications

The framework summarizes the commonalities in the domain of power distribution analysis and is designed as a semi-completed application. The framework users or application developers may extend it to create their own applications. That is, they extend the interfaces and/or classes to create their own sub-interfaces and/or subclasses under the umbrella of the framework. Therefore, the architectural design of the framework is reused by these applications. This section provides an example to illustrate ways to extend and reuse the framework to implement a customized application, which is to optimize the placement of Distributed Resources (DRs) or Distributed Generations (DGs) [16], [19] in a radial distribution system to minimize the system power losses. The example in this section consists of the following five parts:

1) Create a subclass of *AbsCmp* to model the circuit equations for DRs.
2) Reuse and extend *FwdTreeTrav* to provide methods to find the next available DR component in the system.
3) Reuse and extend *TreeLoadFlow* to model load flow algorithm with the considerations of DRs as PV buses.
4) Create a class to model the simulated annealing algorithm that optimizes the DR placement for the minimal power losses.
5) Discuss how to apply *DistributedAlg* layer of the framework to implement the parallel processing of the load flow with DRs and the simulated annealing application.

Interaction diagrams of the Unified Modeling Language (UML) [20] are used to illustrate the interactions between the internal modules of the framework and the external classes created by users.

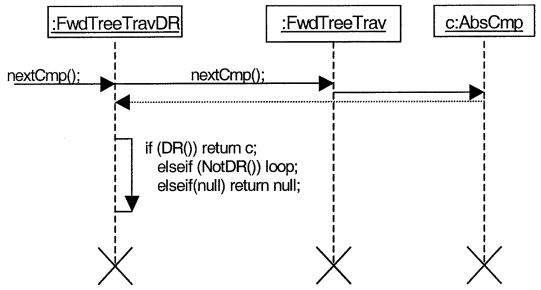

Fig. 11. Interaction diagram illustrating the process of the *FwdTreeTravDR.nextCmp()* method.

A. Modeling DRs

The three interfaces of *AbsCmp* class provide flexibilities to model the essential behaviors of a power distribution component. Hence, DRs can be modeled as a subclass of *AbsCmp*, *DistRes*. The *DistRes* class can be viewed as a unified part of the *Component* layer. In other words, the boundary of the *Component* layer is expanded to handle other components.

B. Finding the Next DR Component

This part discusses a way to model a traverser class to find the next available DR component in a system starting from the root component (substation). This can be implemented by creating a new class, *FwdTreeTravDR*, which is extended from *FwdTreeTrav* class in the *Iterator* layer. The new class has an overridden method *nextCmp()* that has the following activities.

1) Call *super.nextCmp()*, i.e., *FwdTreeTrav.nextCmp()* which returns the first available downstream component, c.
2) If c is an instance of *DistRes* class, return it.
3) If c is not a *DistRes* object, go to step 1.
4) If c is null (no more component in the system), then return null.

As the above steps show, the *FwdTreeTravDR.nextCmp()* method returns the next *DistRes* component, as opposed to a general component that *FwdTreeTrav.nextCmp()* returns. Certainly, the *FwdTreeTravDR* needs to implement the *firstCmp()* method that should return the first DR component, and the other methods defined in *ITraverser* interface in similar ways. Fig. 11 illustrates the interactions that occur when an object of type *FwdTreeTravDR* receives the message *nextCmp()*.

This example also shows the advantages of the Iterator pattern, which separates the graph structural information and traversal methods. When a new traverser is needed, only a new traverser class is created, and there is no need to access classes related to the graph structure.

C. Modeling Load Flow With the Consideration of DRs

Due to the radial or weakly meshed nature of distribution systems, load flows are usually carried out by using the Ladder Load Flow algorithm with backward and forward sweeps [13]. This does not handle PV buses. Although DRs can be modeled as negative load for simplicity, a more accurate model like a synchronous machine model or PV bus model may be desirable. When distributed resources are modeled as PV buses, the default load flow algorithms modeled in the framework need to

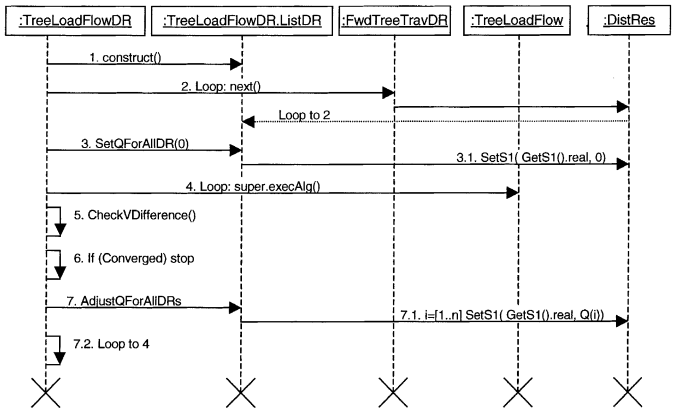

Fig. 12. Interaction diagram for load flow with the consideration of DRs as PV buses.

be extended. The approach, similar to the one in [19], is briefly illustrated as follows. First, set $Q = 0$ for all DRs. Secondly, run a regular load flow to get the V at each DR. Thirdly, check the difference between the calculated V and the predefined V (DRs are PV buses) of all DRs. If not converged, adjust the Q value of each individual DR based on the V difference till V is converged at each DR.

To model the above approach by reusing the framework, *TreeLoadFlowDR* is created as a subclass of *TreeLoadFlow*. The main method in *TreeLoadFlowDR*, *execAlg()*, is briefly described as follows.

1) Construct a list, *ListDR*, to contain all references to DR objects.
2) Traverse the system to obtain all DRs and put them into *ListDR*.
3) Assign $Q = 0$ for all distributed resources and treat DRs as a negative load.
4) Run *super.execAlg()*, i.e., *TreeLoadFlow.exeAlg()*, to calculate the load flow, including the voltages for all DRs.
5) For each DR, compute the difference between the calculated voltage and the pre-defined actual voltage.
6) If the difference is within the tolerance for all DRs, stop.
7) Otherwise, adjust the values of Q for each DR and go to step 4.

Fig. 12 illustrates an interaction diagram of the above approach. As the figure and the above process show, the *TreeLoadFlowDR* class inherits the *TreeLoadFlow.exeAlg()* method for the key calculation in the new load flow with DRs. It also contains a linked-list, *ListDR*, to keep track of all DRs.

D. Optimizing the DR Placement Based on Minimal Losses

In this part the simulated annealing algorithm is applied to optimize the DR placement in a radial distribution system based on minimal power loss. The mathematical nature of this problem is similar to the optimal placement of capacitors. Previous works [21]–[23] presented different algorithms, such as tabu search, simulated annealing, genetic algorithms and others, to solve optimal capacitor placement. Those approaches can be applied to identify the optimal placement of DRs. Here, the simulated annealing approach is selected for demonstrative purpose.

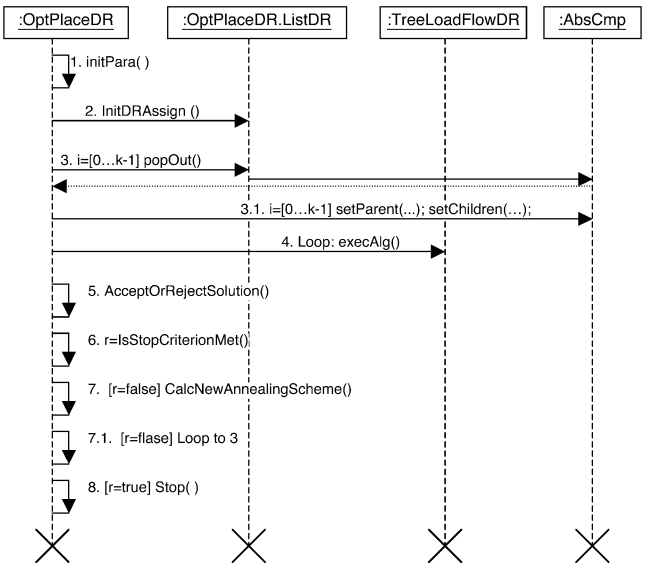

Fig. 13. Interaction diagram illustrating optimal DR placement using simulated annealing.

As previously stated, this paper targets the design extensibility and reusability of the framework rather than a specific algorithm for optimal DR placement based on minimal power losses. As such, details and specifics of the algorithm are not discussed here. However, a high-level design is provided to demonstrate the extensibility of the framework. The design is sketched as follows.

1) Set initial parameters for simulated annealing such as the starting temperature, the annealing rate, etc.
2) Set the initial assignment for DRs.
3) Update the system topology due to the DR locations.
4) Run a load flow with DRs to determine the power loss.
5) Accept or reject the new solution based on solution quality and temperature.
6) Chek whether the stop criterion is statisfied or not.
7) If not, change annealing factor and determine the new DR locations with a perturbation mechanism. Then go to step 3.
8) If yes, stop.

Fig. 13 illustrates how the above algorithm can be implemented in the framework. A class *OptPlaceDR* is created to handle this optimization application. It has a major method *execAlg()* to model the main process of the algorithm. It also contains a linked-list, *ListDR*, to model all available DRs. It should be noted that the *OptPlaceDR.ListDR* list may be a superset of *TreeLoadFlowDR.ListDR*, which is for load flow only. In addition, the *OptPlaceDR* class may contain other assisting methods such as *CalcNewAnnealingScheme()*, etc.

E. Parallel and Distributed Processing

If users only need to implement the application in a standalone machine, the previous discussion is sufficient. However, it may be beneficial if the application is implemented in a parallel or distributed processing scheme, especially for the simulated annealing application that could be very time-consuming for a practical system with tens of thousands of components. In

the following discussion, the design to implement parallel and distributed processing of load flow with DRs is presented first. Then, the parallel and distributed processing of the simulated annealing is briefly described.

Previous work [13] has presented an algorithm for partitioning the load flow calculation so that multiple processors can work together to complete the load flow simultaneously. Here, the high-level design issue is discussed. The design starts from the creation of a concrete subclass of *AbsDLF*, *DTLFDR*, to handle the remote method invocation of radial load flows with DRs. If a reference of *TreeLoadFlowDR* is passed as the first parameter of the *invokeRemoteAlg()* method in the *DTLFDR* class, the remote invocation of *TreeLoadFlowDR* will be automatically executed through polymorphism. For this distributed processing scheme, users need to partition the system first and then send the associated subsystem to the remote execution server.

Once the distributed processing of load flow with DRs is implemented, it can be directly used for the simulated annealing application. A simple, effective approach to achieve the benefit of distributed processing and to minimize the development cost is to utilize the *DTLFDR*, instead of *TreeLoadFlowDR*, to carry out the load flow calculation (Step 4 in the proposed simulated annealing algorithm in Part D). Since the load flow calculation consumes most of the computing power, it is certain that *DTLFDR.exeAlg()* benefits the overall wall-clock running time as opposed to its sequential version, *TreeLoadFlowDR.exeAlg()*.

V. Benefit of Framework

As previously discussed, this framework provides a foundation of the commonalities in power distribution analysis, such as distribution components, topological traverse, and fundamental algorithms. The framework users can build their own, high-level analysis applications based on the framework. The benefits to the users are as follows.

1) **Reduced cost and enhanced reusability**. The top-down dependency among four layers reduces the cost and effort for reusing the framework. Since lower layers do not depend on higher layers, users can always reuse lower layers without the need to obtain higher layers. For instance, if users want to build their own load flow, they need only the *Component* and *Iterator* layers, but not the other two layers. Also, if a user wants to implement a high-level algorithm on a stand-alone machine like the example in Section IV-A–D, only the lower three layers are needed.
2) **Extensible and reusable internal hierarchy**. The internal class diagram of each layer follows a strict hierarchical structure. Hence, it is relatively easy to extend the in-framework classes to implement user-defined, specific classes.
3) **Maintainable framework-based applications**. Since standard design patterns are employed in the framework design, it is easy for users to understand and reuse the design. Then, once a user understands the design of the framework, it will be easy for the user to understand applications developed by others which are based on the framework. Therefore, it is more maintainable for all applications based on the framework.
4) **Interface-driven dependency**. The dependencies among layers are driven by interfaces, so each layer is relatively independent from other layers. In other words, the internal structure of one layer is not visible to other layers. Hence, there is reduced coupling in the framework.
5) **Language-independent interfaces**. The interfaces of each layer are designed in IDL, which is a standard interface definition language supported by many programming languages. Therefore, the framework can be essentially accessed by applications in heterogeneous languages.

VI. Summary

Frameworks are a promising technology to reuse design for an entire domain to reduce cost and improve software quality. This paper presents an architectural design for a framework, which aims to summarize the commonalities in software design for power distribution systems, including components, topological traverses, and analysis algorithms. It also supports the idea of distributed computations.

The architecture of the framework is layer-driven with strict top-down dependencies to reduce software coupling and to gain reusability and extensibility. Standard design patterns are applied to the layers to achieve maintainability. In addition, dependencies are carried out through IDL interfaces to further reduce software couplings. The language-independent IDL interfaces make the framework essentially callable by other heterogeneous languages.

References

[1] J. Zhu and P. Jossman, "Application of design patterns for object-oriented modeling of power systems," *IEEE Trans. Power Syst.*, vol. 14, pp. 532–537, May 1999.
[2] J. Zhu and D. Lubkeman, "Object-oriented development of software system for power system simulations," *IEEE Trans. Power Syst.*, vol. 12, pp. 1002–1007, May 1997.
[3] R. P. Broadwater *et al.*, "Application programmer interface for the EPRI distribution engineering workstation," *IEEE Trans. Power Syst.*, vol. 10, pp. 499–505, Feb. 1995.
[4] S. Pandit, S. Soman, and S. A. Khaparde, "Object-oriented design for power system applications," *IEEE Computer Applicat. Power*, vol. 13, pp. 43–47, Oct. 2000.
[5] R. P. Broadwater, M. Dilek, and J. Thompson, "Centralized, distributed responsibility, and decoupled object-oriented software designs for power systems," in *Proc. IEEE Power Eng. Soc. Summer Meeting*, vol. 2, 2001, pp. 1025–1028.
[6] X. G. Wei, Z. Sumic, and S. S. Venkata, "ADSM—an automated distribution system modeling tool for engineering analysis," *IEEE Trans. Power Syst.*, vol. 10, pp. 393–399, Feb. 1995.
[7] E. Gamma *et al.*, *Design Patterns—Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1997.
[8] *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, M. E. Fayad *et al.*, Eds., Wiley, New York, 1999.
[9] *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, M. E. Fayad *et al.*, Eds., Wiley, New York, 1999.
[10] J. Lakes, *Large-Scale C++ Software Design*. Reading, MA: Addison-Wesley, 1996.
[11] B. H. Kim and R. Baldick, "A comparison of distributed optimal power flow algorithms," *IEEE Trans. Power Syst.*, vol. 15, pp. 599–604, May 2000.

[12] C. L. Borges *et al.*, "Composite reliability evaluation by sequential Monte Carlo simulation on parallel and distributed processing environments," *IEEE Trans. Power Syst.*, vol. 16, pp. 203–209, May 2001.
[13] F. Li and R. P. Broadwater, "Distributed algorithms with theoretic scalability analysis of radial and looped load flows for power distribution systems," *Elect. Power Syst. Res.*, vol. 65, no. 2, pp. 169–177.
[14] R. P. Broadwater, J. C. Thompson, and T. E. McDermott, "Pointers and linked lists in electric power distribution circuit analysis," in *Proc. Power Ind. Comput. Applicat. Conf.*, 1991, pp. 16–21.
[15] A. Losi and M. Russo, "An object oriented approach to load flow in distribution systems," in *Proc. IEEE Power Eng. Soc. Summer Meeting*, vol. 4, 2000, pp. 2332–2337.
[16] R. E. Brown, *Electric Power Distribution Reliability*. New York: Marcel Dekker, 2002.
[17] ——, "Distribution reliability assessment and reconfiguration optimization," in *Proc. IEEE Power Eng. Soc. Transm. Dist. Conf. Expo.* , vol. 2, 2001, pp. 994–999.
[18] M. Gudgin, *Essential IDL: Interface Design for COM (The Develop-Mentor Series)*. Reading, MA: Addison-Wesley, 2000.
[19] F. Li, R. Broadwater, J. Thompson, and F. Goodman, "Analysis of distributed resources operating in unbalanced distribution circuits," in *Proc. IEEE-Power Eng. Soc. Summer Meeting*, vol. 4, 2000, pp. 2315–2319.
[20] R. Pooley and P. Stevens, *Using UML Software Engineering With Objects and Components*. Reading, MA: Addison-Wesley, 1999.
[21] H. D. Chiang *et al.*, "Optimal capacitor placements, replacement, and control in large-scale unbalanced distribution systems: system modeling and a new formulation," *IEEE Trans. Power Syst.*, vol. 10, pp. 356–362, Feb. 1995.
[22] D. Jiang and R. Baldick, "Optimal electric distribution system switch reconfiguration and capacitor control," *IEEE Trans. Power Syst.*, vol. 11, no. 2, pp. 890–897, May 1996.
[23] R. A. Gallego, A. J. Monticelli, and R. Romero, "Optimal capacitor placement in radial distribution networks," *IEEE Trans. Power Syst.*, vol. 16, pp. 630–637, Nov. 2001.

Fangxing Li (S'98–M'01) received the B.S. and M.S. degrees in electric power engineering from Southeast University, Nanjing, China, in 1994 and 1997, respectively. He received the Ph.D. degree in computer engineering from Virginia Tech, Blacksburg, in 2001.

Currently, he is a Senior Consulting R&D engineer at ABB Consulting, Raleigh, NC. His research interests include computer applications in power systems, power distribution analysis, energy market simulation, and continuous and discrete optimization.

Robert P. Broadwater (M'71) is a Power Systems and Software Engineering Professor at Virginia Tech, Blacksburg, where he teaches courses in applied software engineering and large-scale software development.

He works in the area of computer-aided engineering for electrical distribution system analysis, design, and operations.