CORBA-Based Object-Oriented Framework for Distributed System Design*

Nestor Michelena, Christopher Scheffer,† Ryan Fellini, and Panos Papalambros

AUTOMOTIVE RESEARCH CENTER
COLLEGE OF ENGINEERING
THE UNIVERSITY OF MICHIGAN
2250 GG BROWN BLDG.
ANN ARBOR, MI 48109-2125

ABSTRACT

A generic framework for designing large, complicated systems is formulated, implemented, and used in the design of mechanical systems, including a pressure vessel, an automotive hybrid powertrain, and a tracked vehicle. The framework supports simulation-based design, distributed and heterogeneous computing resources, custom and legacy simulation and analysis codes, reconfigurability of the design problem, and security of operation across untrusted networks. The framework also facilitates the implementation of methodologies for system design that employ design model partitioning and coordination, as well as a variety of models and search algorithms. Common Object Request Broker Architecture (CORBA) middleware for distributed, object-oriented applications was selected to develop and implement the framework. Framework components include subsystem model, design model, search engine, design model partitioning, design coordination, and user interface.

*Communicated by E. J. Haug
†Member of the Technical Staff, TASC, Reading, MA.
1. INTRODUCTION

Computer hardware, middleware, and engineering system simulation and analysis software ("simulations," for short) have reached the level of performance and development needed for implementation of practical simulation-based tools for system design. Although the importance of automated design tools is indisputable for parametric design, these tools are also critical to other more creative stages of the design process. Well-conceived and efficient parametric design tools play a major role in the generation and validation of design concepts and configurations. For example, design tools that use mathematical programming algorithms to search the design space are routinely used for conceptual (topology) design of structures and mechanisms [1,2].

Past limitations in hardware and software have resulted in a proliferation of automated design tools that either (1) focus on single system components that, when put together, result in a suboptimal system design, or (2) use low fidelity system models and simulations to produce designs that need substantial refinement. Recent advances in hardware, middleware, and software require new design paradigms, methodologies, and computational environments to bridge the gap between component design and system capabilities. In automotive applications, for example, although high fidelity models and simulations of the powertrain subsystem and vehicle dynamics are readily available, prevailing practice tends to isolate the design of individual subsystems by oversimplifying their interactions. They often assume torque vs. time and speed vs. time simulation inputs, instead of using the natural torque-speed interface between vehicle dynamics and propulsion models.

Current design and computational environments typically treat simulation and analysis software as stand-alone tools, conceived for component or subsystem analysis and executed on a single computing platform. A simulation is designed for a single iteration, instead of several consecutive ones needed in a design optimization study. Integration with other commercial/off-the-shelf software or with automated design tools is hardwired by function calls or parsing and writing text files. System design requires the integration of several subsystem simulations to evaluate system characteristics with design space search capabilities.

In this paper, an object-oriented distributed design framework view [3] is presented with the following characteristics:

1. Platform independence: support of distributed, heterogeneous, and reconfigurable computing resources.
2. Multiple levels of model abstraction and fidelity: support of distributed search algorithms with model hierarchies required in decomposition and coordination strategies.
3. Reconfigurability of the design problem: plug and play of models, data-
bases, and simulation and synthesis tools to allow use of progressively more complex analytical models.

4. Accommodation of both custom-developed and legacy (existing) codes.

5. Adequate time performance for design purposes: although real-time calculation is not necessary for design, the computational time (including model preparation) should not add significantly to design cycle time, and speedups from distributed computing should not be overshadowed by network communication overheads.

6. Security of operation across untrusted networks: analysis tools should be developed and maintained at different locations while still being used to solve a system problem.

The distributed design framework facilitates implementation of a methodology for large-scale system design that builds on previous work on design model partitioning [4–6] and coordination [7,8]. The distributed design framework was originally proposed by Papalambros et al. [9].

II. DISTRIBUTED SYSTEM DESIGN

Consider the design of the tracked vehicle shown in Fig. 1. The existence of simulations for the hull, powertrain, and upper and lower tracks and databases

![Diagram of tracked vehicle with subsystems and models.]
describing several environment and terrain scenarios is assumed. Simulations for each subsystem have different fidelity and are implemented on a heterogeneous computing environment. Some subsystem models are implemented as lengthy analysis codes in FORTRAN or MATLAB/SIMULINK, others use interpolated lookup tables, and some models are used in conjunction with commercial packages such as DADS and NASTRAN. Subsystem models may consist of other submodels (e.g., the powertrain subsystem may include a gas turbine or diesel engine, torque converters, a transmission, and a differential). Selection of appropriate subsystem simulations depends on the type of system characteristics that need to be predicted (e.g., fuel economy, performance, vehicle dynamics, or track vibration) to formulate a goal-oriented decision problem of the following form: \textit{Find the values of the design parameters }x\textit{ that improve the design criteria }f(x)\textit{.}

Design of a large-scale system such as a tracked vehicle requires integration of subsystem simulations with a search engine that drives the design process. Component simulations are typically developed by independent groups of dispersed people. This issue is addressed as a problem of software interoperability within a distributed object-oriented framework, as described in Section II.A.

A second issue is the synthesis process itself; i.e., the design search engine. Formal design optimization methods (based on global and local search algorithms) are highly successful when used in conjunction with relatively small, well-behaved, single-discipline subsystem models. Multidisciplinary design optimization (MDO) practitioners, however, face high computational cost and lack of robustness of distributed optimization algorithms, in addition to the problem of interoperability between simulations and search engines. This second issue is addressed by using a design methodology based on model partitioning and coordination, hybrid system models and search algorithms, and distributed computation, as described in Section II.B.

A. CORBA Environment for the Interoperability of Simulation, Analysis, and Design Software

The Common Object Request Broker Architecture (CORBA) [10] has been selected as middleware for building a distributed design framework whose architecture is presented in Section III. CORBA is an industry standard for distributed, heterogeneous, object-oriented applications. It is open, robust, interoperable, multiplatform, and multivendor supported. CORBA uses the Interface Definition Language (IDL) to specify attributes, operations, and parameters of each operation that a server provides in the implementation of a given object. CORBA objects can be accessed by remote clients via method invocations. Both the language and the compiler used to create server objects are totally transparent to clients. Clients do not need to know where the distributed object resides,
what operating system it executes on, and how the server object is implemented. Clients only need to know the interface their server objects publish.

In this context, examples of server and client objects are subsystem simulations and design search engines, respectively. The Internet Inter-ORB Protocol (IIOP) enables the interconnection of large distributed applications across the Internet. Legacy software can be integrated into a CORBA environment by stripping off any I/O from the source code and creating a library of (simulation and analysis) methods that can be called from C or C++ code. A CORBA-IDL interface enables entering data into the remote object, executing defined functions, and retrieving data. Commercial software can also be wrapped with a CORBA interface.

B. Design Methodology Based on Model Partitioning, Coordination, Hybrid Models and Algorithms, and Distributed Computing

Modern engineering design of even a simple component requires using mathematical models to predict the component’s behavior and to select a design whose value is considered satisfactory. A typical design approach consists in formulating an optimization problem, using a predictive model to estimate the design criterion and constraint functions, and applying formal methods to search the design space for a point or points that improve the design criterion. This approach, although easy to conceive, presents several challenges to a system design team, given the following characteristics of system design problems:

1. Levels of accuracy and resolution of mathematical models may vary across the design space. In general, the more accurate the models, the longer the analysis and simulation runtimes. A single simulation run may take hours to complete.
2. Proper system design criteria and constraints are difficult to identify, given the multiobjective nature of a system design problem. It is often not clear whether a system characteristic should be treated as a design criterion (objective function), a design constraint, or both.
3. Several models are needed to predict system characteristics.
4. Noisy and/or discontinuous performance responses are prevalent in simulations that contain empirical data (e.g., lookup tables or maps), numerical integration, or discrete decisions.

In summary, system design problems are characterized as follows:

- High computational cost
- Difficult design problem formulation
Multiple subsystem simulations, analyses and models
Nonsmooth, noncontinuous models

Most of these issues are addressed here by using the design methodology depicted in Fig. 2 and supported by the proposed object-oriented distributed design framework. Description of the major components of the design methodology (in the white boxes in Fig. 2) follows.

1. Design model partitioning and coordination. Standard search techniques cannot deal well with the difficulties enumerated in the previous section. Gradient-based local optimization methods are successful when applied to relatively small, smooth design models. Direct local and global search methods are suitable only for computationally inexpensive models.

Distributed design, by means of model partitioning and coordination, takes advantage of the partitioned design model by solving smaller design problems. The same level subproblems may be solved in parallel, using the optimization technique most suitable for the underlying submodel, gaining in robustness, speed, and engineering interpretation of results. Coordination strategies, such as sequentially decomposed (SD) programming [7] or hierarchical overlapping coordination (HOC) [8], ensure convergence to the system design solution.

Distributed design requires partitioning of the original system design model into submodels and/or clustering of an already partitioned system model. Although engineering insight may help in recognizing clusters of subsystem models, the decision may be difficult for a large system. System optimization cost depends on how loosely the individual subproblems are coupled by linking or coupling variables. Model cluster identification should also consider the avail-

![Diagram](image-url)  
**Fig. 2.** Methodology for system design based on design model partitioning and coordination, hybrid system models and search algorithms, and distributed computation.
ability and throughput of computational resources and the cost of performing individual analyses. Model partitioning can be formulated as a high-level optimization problem and solved using graph partitioning [4,5] or integer programming algorithms [6]. Web-based implementations of these approaches can be found at the following URLs: http://arc.engin.umich.edu/graph_part.html and http://arc.engin.umich.edu/ilp_part.html.

2. Distributed computation. Distributing computation across a number of networked workstations is a simple, cost-effective way of speeding design space search. Computation can be distributed at the analysis level (i.e., for the execution of simulations) or at the synthesis level (i.e., for the design search itself). At the analysis level, parallelization can happen at the program level (DO or FOR loops), at the job level by the operating system (multithreading), or by coarse grain parallelization (domain decomposition or substructuring). At the synthesis level, computation can be distributed for estimation of sensitivities, evaluation of design criteria and constraints, or design model decomposition and coordination. The object-oriented design framework presented in Section III enables distributed computation at the synthesis level, and by coarse grain parallelization.

3. Hybrid models and algorithms. Synthesis approaches based on a single search algorithm perform well when the underlying design model is inexpensive to evaluate or is well behaved. Large-system design requires using a variety of system models and search algorithms. Low fidelity models (e.g., surrogate models) should be used for the first few iterations, whereas high fidelity models should only be used to refine the search. For hybrid search approaches, low fidelity models are more adequate for the global search stage (using, for example, evolutionary or simulated annealing algorithms), whereas high fidelity models work better during the local search stage (using a nonlinear programming algorithm). An object-oriented design environment facilitates such implementations, since a plug-and-play capability allows interchanging models and optimization algorithms without reintegration of the software system.

III. OBJECT-ORIENTED DISTRIBUTED SYSTEM DESIGN FRAMEWORK

A system design framework has been developed and implemented based on the CORBA standard. The framework specifies formal interfaces between several types of distributed components. The components include subsystem models, design models, search engines, design model partitioning, design coordination, and user interface, as shown in Fig. 3. The diagram shows the possible

1Object model diagrams follow the object model notation of Rumbaugh et al. [11].
multiplicity of search engines, simulation and analysis models, and design submodels for a given design problem. Design submodels may have been generated by the model partitioning component.

Individual components can be implemented using libraries of fundamental mathematical objects such as vectors and matrices. Examples of these libraries are the Hilbert Class Library (HCL) [12] and the Template Numerical Toolkit (TNT) [13]. Each component can have several distinct implementations that are completely interchangeable. Since each component is a CORBA object, they can reside on distributed computers in separate executables. They are selected and bound at runtime to dynamically create a custom design environment.

The user of the framework selects the subsystem models needed to describe the system's characteristics. Based on inputs and outputs of these models and computational requirements for their evaluation, the model partitioning component will define one or more design models corresponding to design subproblems. Each design subproblem will be solved by a search component. The user may select different search engines for each design subproblem. The design of the system will be determined by a coordination component, using either hierarchical or nonhierarchical coordination strategies. The selection and customization of design model components is performed through the user interface. This component may also be interchanged among various types including batch, interactive, and graphical interfaces.

We will compare the proposed design framework with two object-oriented class libraries for optimization; namely, the CWP Object-Oriented Optimization Library (COOOL) [14] and OPT++ [15]. Three major differences are that these libraries do not support (1) constrained optimization, (2) distributed optimization

\(^2\)CORBA objects are bound when communication channels exist among them.
based on model partitioning and coordination, and (3) distributed computing based on CORBA objects.

Another CORBA-based integration infrastructure is DITools [16], an evolution of TACTICS [17]. DITools is used to configure system simulations from distributed subsystem models. As in our framework, it provides a modeling API through which legacy codes can be integrated. It also provides a graphical interface through which system simulations are constructed, initialized, and controlled. Both DITools and our framework allow the selection of distributed objects and the dynamic creation of system models. DITools provides the capability of dynamically discovering CORBA interfaces of individual models. One difference is that DITools does not explicitly provide support for optimization components such as design models or search engines.

A. Subsystem Model Component

Subsystem model components (SMC) encapsulate subsystem simulations as CORBA objects. Given an input, these components generate output of interest to other model components, the search engine and design coordination components. A system is modeled as a set of coupled analyses, as shown in Fig. 4. Each simulation model is associated with a set of independent local input variables \(x_i\) and a set of output variables \(y_i\) that are calculated by the analysis. Coupling between models occurs when output variables of one analysis are used as input to another. These variables are the coupling variables \(y_{ij}\). Coupled models are also likely to share common inputs, called linking variables \(x\). SMCs have postprocessing capabilities to produce output that can be used by the search engine(s) or the design coordination component to formulate design criteria and constraints.

![Diagram](image-url)  

**Fig. 4.** A system is modeled as a set of coupled analyses.
B. Design Model Component

Design model components (DMC) are interfaces between subsystem model components and the search engine and design coordination components. DMCs allow definition of design problems as optimization problems of the following form:

\[
\text{Find } x \in X \subseteq \mathbb{R}^n \text{ such that} \\
f(x) \text{ is improved and} \\
x_l \leq x \leq x_u \\
g_l \leq g(x) \leq g_u \\
h(x) = h_0
\]

(1)

where \( x \) is the vector of design variables, \( f(\cdot) \) is a vector function representing design criteria to be improved, and \( g(\cdot) \) and \( h(\cdot) \) are vector functions representing design constraints to be satisfied. The simple (upper and lower) bounds on variables and functions, \( x_u, g_u, x_l, \) and \( g_l \), are fixed during optimization. In general, any functions of subsystem model input and output variables may be used to construct a problem of this type. The design model component accepts a vector of design variables, maps this vector to subsystem model inputs, runs the necessary analyses, and maps I/O data to objective and constraint values. The design model component can run single or multiple simulations, or a subset of a simulation. This effectively separates the design problem definition from analyses. The user is free to formulate problems to handle coupling and linking variables. For example, the coupling variables may be treated as independent design variables at the subproblem level, with a coupling equality imposed at the system level. Alternatively, linking variables may be treated as parameters at the subproblem level and as design variables at the system level.

Figure 5 shows the design model class made up of design criteria (f), constraints (g, h, and simple bounds), and design space (x and set constraint) classes. These classes provide the search engine and design coordination components with methods to set the value of the design variables and to retrieve the values of criteria and constraint functions. Mapping between simulation I/O and design variables, constraints, and criteria is defined after the objects are created and bound.

A design model object is instantiated for each design subproblem and dictated by model partitioning. Since a simulation typically evaluates design criteria and constraint functions simultaneously, the design model class provides methods that launch the simulation and update design criteria and constraints. These methods can launch a simulation multiple times to evaluate gradients. The de-
Fig. 5. Object model of design model component.
sign space object maintains the integrity of the design vector. A simulation, for example, may not accept a noninteger number as one of its inputs. Lagrange multipliers are attributes in the constraint subclasses. The design model component contains subcomponents to estimate first and second order information of the design criteria and constraints. This feature provides the search engine with the flexibility to switch or reconfigure gradient or Hessian estimators on the fly.

Comparatively, the COOOL library defines two classes that would correspond to the design model component: the Model class and the ObjectiveFunction class. The former class includes information about the design vector. Predefined objective functions such as the Rosenbrock, quadratic, and min-norm fit functions are derived subclasses. There is a separate class called the SlaveForward class which is used when an independent executable process (the slave) is needed to evaluate the objective function. A SlaveForward object sends trial design vectors to the standard input of the slave process and receives data values from the standard output of the slave. A \texttt{gradient()} method is part of the ObjectiveFunction class, but it returns a NULL pointer when analytical derivatives are not available. In this case, estimation of derivatives by finite differences is done within the optimization object.

\texttt{OPT++} contains three nonlinear problem classes (\texttt{NLP0}, \texttt{NLP1}, \texttt{NLP2}) to account for the availability of first and second order analytical derivatives. All three classes contain information on the problem dimension, current point, and objective function value, as well as a method to evaluate the objective function. \texttt{NLP1} is derived from \texttt{NLP0} by adding a member for the objective function gradient and a method to evaluate the gradient. Similarly, \texttt{NLP2} is derived from \texttt{NLP1}. \texttt{OPT++} has no provision to run slave processes to evaluate the objective function.

### C. Model Partitioning Component

The model partitioning component (MPC) may be used to identify and define multiple subproblems acting on one or more subsystem models. Based on a system design model definition, the partitioning component determines how each design relation depends on inputs and outputs of the subsystem models. This can be calculated by examining the coefficient matrix in a linear model or by performing finite difference sensitivity evaluations in nonlinear models. Any nonzero element in the coefficient or sensitivity matrix indicates a dependence. The term design relation is very general and may correspond to an algebraic or differential equation, a discretized continuum equation, a response surface, or a simulation used to evaluate state/behavior variables. A black-box simulation may be treated as a single design relation or as a collection of relations, each one corresponding to an output from the simulation.

Using model dependence information, along with the desired number of sub-
problems, partitioning algorithms determine how subsystem inputs and outputs map to subproblem definitions. For example, a powertrain system may be composed of engine, driveline, and overall powertrain simulations, each implemented in different simulations. The user may wish to partition this system into two subproblems. After calculating dependencies, the partitioning component defines the first subproblem to consist of the entire engine subsystem and the subset of the powertrain subsystem related to the engine. The second subproblem may consist of the driveline and the remaining powertrain relations.

D. Search Component

The search engine component (SEC) implements methods to be applied to a design model object to search for a design vector that improves the design criteria and satisfies the design constraints. The framework considers three types of search methods: global search, derivative-free local search, and gradient-based local search. Derivative-free and global search methods are explained in detail by Fellini [18]. An object-oriented framework allows using different search methods for the same design model, with very little coding effort. With several types of search algorithms, the user can choose a particular algorithm that works well for the given design problem or subproblem. For example, a convex approximation algorithm may be used for structural design, a branch and bound algorithm may be used for problems that involve discrete variables, or a genetic algorithm may be used for problems in which a global minimum is required. All three algorithms may need to be combined under a coordination strategy to solve a system design problem.

Existing optimization software can be wrapped as CORBA objects without modification to the original code. However, an object-oriented design of the search algorithms enables code reuse in implementation of new algorithms and facilitates code maintenance. Hybrid search strategies can be easily implemented if the individual algorithms are based on a uniform class structure, such as that shown in Fig. 6. Software to solve quadratic programming problems or line searches, for example, can be directly reused to implement optimization software for general nonlinear programming.

The COOOL library provides three local, unconstrained optimization methods (conjugate gradient, Powell's search, and nonlinear simplex), and two global optimization methods (simulated annealing and genetic algorithms). OPT++ contains a direct local search method (parallel direct search); two classes of unconstrained, gradient-based methods (conjugate gradient and Newton like

\textsuperscript{3}CORBA's Dynamic Invocation Interface (DII) allows discovery of methods to be invoked at runtime, although at a substantial programming effort.
methods); and three bound-constrained methods (ellipsoid, barrier, and active-set Newton methods).

E. Design Coordination Component

Whenever the design partitioning component dictates breaking down the system design problem into multiple subproblems, a coordination strategy is needed to achieve a system optimum by solving the individual subproblems. Coordination strategies are provided by the design coordination component (DCC). Typically, a system design problem is broken down by analysis—one subproblem per analysis model with a coordination problem acting on the entire system. However, in this framework, the system design model can be partitioned into subproblems containing any arbitrary combination of simulations. A generic coordination object has been implemented that can be configured for several types of hierarchical or nonhierarchical model partitioning. It stores design and state information for the entire system. Subproblems are instructed to call back to the coordinator for updated system data. These callbacks can occur at each step of the subproblem or after convergence.

An example of a strategy that can be used with this coordination object is (hierarchical) sequentially decomposed (SD) programming [7]. In this formulation, a master problem is configured to act on the entire design model. At each iteration of the master problem optimization, subproblems are given linking variable information to be used as parameters as they are optimized with respect to their local design variables. Subproblems are optimized at each system design iteration until the system converges to a near optimum.

A coordination strategy similar to Sobieski’s linear decomposition method [19] can be implemented by reformulating the master problem to act only on linking variables. Strategies for nonhierarchical model partitioning can be created by optimizing each subproblem with respect to local variables while approximating constraints present in other subproblems, as in concurrent subspace optimization (CSSO) [20].

Hierarchical overlapping coordination (HOC) [8] is a coordination strategy that simultaneously uses two or more different decompositions of the design model. There is no master problem. Coordination is achieved by the exchange of information between decompositions. Linking and coupling variables for one of the decompositions are fixed at values that result from the solution of a number of independent subproblems associated with a second decomposition.

---

Fig. 6. Object model of search component.
F. User Interface Component

The user interface component (UIC) is the only component with which the user interacts. It transfers all user requests to the appropriate components. It is responsible for initially binding and configuring the other components. Several types of user interfaces may exist, depending on user preferences. An interface component may simply be hard-coded with instructions to the other framework components. Another interface may read a text file of instructions and execute them in batch. Users not familiar with the framework may prefer a graphical user interface (GUI) that prompts the user for needed information. Custom graphical interfaces may be developed for any platform. CORBA allows a Java-based GUI to be downloaded and executed on any computer with a Java-enabled web browser.

IV. IMPLEMENTATION ISSUES

Several components described above have been implemented in C++ and IDL. A design model component has been created that allows design criteria and constraints to be defined as linear combinations of subsystem model inputs and outputs. A generic coordination component has been implemented. It is designed for nonhierarchical coordination strategies, but may be customized by the user for hierarchical strategies. A simple user interface component has been developed in which the user directly codes information specific to the design problem. Other implementations can be added to the framework for each of these component types.

Several issues concerning modularity and distributed operation related to the CORBA implementation have been addressed in creation of the framework. Components have been designed to have knowledge only of the interfaces and basic responsibilities of each other. To be considered a component, an object must implement all its interfaces. These interfaces are defined as generically as possible, so that many types of codes may be integrated as components. For example, the interface definition (written in IDL) for the search component is

```idl
// IDL code for search component
interface idlSearchEngine {
    oneway void ParallelRun (in vectord x0); // Starts search in parallel
    void Run (in vectord x0); // Starts search
    void setModel (in idlDesignModel mod); // Identify design model
};
```

All data in CORBA are passed by value, and all data types must be defined in IDL. Specific objects are not defined in IDL and cannot be passed by value. CORBA object references are used to pass information about the location of
distributed objects. Data of constructed types are passed using IDL-defined structures. The above IDL code shows the passing of a CORBA object reference (idlDesignModel) and an IDL data structure (vectord). The latter parameter corresponds to the starting point in the search.

To operate as efficiently as possible in a distributed environment, intercomponent data transfer is minimized and parallel execution is used. Coarse-grained objects are defined to simplify communication. Parallel execution is implemented through use of nonblocking (one-way) calls for starting multiple subsystem model and design model components. The controlling (client) object instructs each (server) component to begin execution. The components operate in parallel and call back to the controller when finished. An example of callback, in which a subsystem model executes its analysis code and returns its results, is

```c++
// C++, implementation of a subsystem model ‘SSM1’
void SSM1::Run (const vectord& x, oenv& e) {
    // Evaluate model at x and assign results to vector y
    ...
    y[1] = ...
    y[2] = ...
    // make call back to model
    model -> setResult (y);
}
```

Orbix software [21] is used for CORBA implementation and Orbix proprietary protocol is used to dynamically bind objects. This is simpler than IIOP, but can only be used with Orbix objects. Below is an example of an Orbix binding call in the user interface code. Note that only two character strings representing the component name (:CFSQP) and the remote host computer (phaethon) are needed.

```c++
// C++, bind to search component from the user interface component
idlSearchEngine_var search;
search = idlSearchEngine: :bind (":CFSQP", "phaethon");
```

For simplicity, static interface invocations are used; i.e., all components are coded with knowledge of each other’s interface definitions. Because of the explicit interface knowledge, direct peer-to-peer communication is easily implemented. Any object can be both a client and a server. In the following sample code, a design model relays requests from a search component to each subsystem model component:

```c++
// C++, implementation of design model ‘DM1’
void DM1::Update (const vectord& x, oenv& e) {
    // update inputs based on local design variable values
```
... // run each subsystem analysis
    for (i = 1; i <= na; i++)
      if (analysis[i]) analysis[i] → Run (input[i]);
    return;
}

It is desirable to reuse existing code as much as possible, because it has usually been debugged and can be treated as a black box. Existing code can be wrapped as a CORBA object if it is in callable form from C++ or any other CORBA-supported language. CFSQP [22], a sequential quadratic programming algorithm written in C, has been wrapped as a search component, without any modification to the existing code, as follows:

    // C++, implementation of search engine 'CFSQP'
    void CFSQP: :Run (const vector& x0, cenv& c) {
      // set up initial data

      // make call to external C function
      cfsqp (argument list . . . );
      // put results into form of IDL vector struct

      // instruct design model to call back to controller
      model → Finished (x);
      return;
    }

V. ILLUSTRATIVE EXAMPLES

A chemical pressure vessel, an automotive hybrid powertrain, and a tracked vehicle are used as illustrative examples of the distributed, object-oriented design framework.

A. Coordination Strategies for Distributed Design

To demonstrate the use of coordination strategies for distributed design, SD programming [7] is used to solve a pressure vessel design problem taken from Beightler and Phillips [23]. The problem uses material cost as objective function, with 1962 ASME Boiler and Pressure Vessel Code [24] design relations as constraints. The optimization problem statement is as follows:

Minimize $f_1 + f_2 + f_3 + f_4$
where
\[ f_1 = 0.662 \, R \, t_s L \quad \text{(shell cost)} \]
\[ f_2 = 1.777 \, R^2 t_h \quad \text{(head cost)} \]
\[ f_3 = 1.582 \, t_s^2 L \quad \text{(shell welding cost)} \]
\[ f_4 = 19.84 \, R t_s^2 \quad \text{(head welding cost)} \]
subject to
\[ g_1: 0.131 \, R / t_h - 1 \leq 0 \]
\[ g_2: 0.0193 \, R / t_s - 1 \leq 0 \]
\[ g_3: 413000 / R^2 L - 1 \leq 0 \]
\[ g_4: 0.00417 \, L - 1 \leq 0 \]
\[ g_5: 0.1 - R \leq 0 \]
\[ g_6: 0.1 - L \leq 0 \]
\[ g_7: 0.1 - t_s \leq 0 \]
\[ g_8: 0.1 - t_h \leq 0 \]

where \( R \) and \( L \) are the radius and the length of the cylindrical shell, respectively; \( t_s \) is thickness of the cylindrical shell wall; and \( t_h \) is thickness of the spherical heads. The problem is partitioned into one master problem and two subproblems. The subproblems are solved with respect to local variables, \( t_s \) and \( t_h \); whereas the master problem acts on the linking variables, \( R \) and \( L \), as well as on the local variables. Optimal values of linking variables determined by the master problem are fixed for the subproblems. The design point is updated every time the master problem or the subproblems are solved. The separate problem statements are as follows:

**Master problem**

Minimize \( f(R, L, t_s, t_h) := 0.662 \, R t_s L + 1.777 \, R^2 t_h + 1.582 \, t_s^2 L + 19.84 \, R t_s^2 \)
subject to
\[ g_1(R, t_h): 0.131 \, R / t_h - 1 \leq 0 \]
\[ g_2(R, t_s): 0.0193 \, R / t_s - 1 \leq 0 \]
\[ g_3(R, L): 413000 / R^2 L - 1 \leq 0 \]
\[ g_4(L): 0.00417 \, L - 1 \leq 0 \]
\[ g_5(R): 0.1 - R \leq 0 \]
\[ g_6(L): 0.1 - L \leq 0 \]
\[ g_7(t_s): 0.1 - t_s \leq 0 \]
\[ g_8(t_h): 0.1 - t_h \leq 0 \]

**Subproblem 1**

Minimize \( f(t_s) := 0.662 \, RL t_s + (19.84 \, R + 1.582 \, L) \, t_s^2 \)
subject to
\[ g_2(t_s): 0.0193 \, R / t_s - 1 \leq 0 \]
\[ g_7(t_s): 0.1 - t_s \leq 0 \]
Subproblem 2

Minimize \( f(t_h) = 1.777 \ R^2 t_h \)
subject to
\[
\begin{align*}
g_1(t_h) & : 0.131 \ R/t_h - 1 \leq 0 \\
g_3(t_h) & : 0.1 - t_h \leq 0
\end{align*}
\]

The subsystem model object PressureVessel is created to evaluate the objective and constraint functions of the entire problem. This object takes the four design variables as input. The design coordination component contains the logic of the SD algorithm. Two instantiations of the design model object are used for the subproblems. The use of these two subproblems enables running the optimization in parallel and on separate computers for computational speedup.

To demonstrate the use of different search engine components on problems residing on different computers, the SQP gradient-based optimization algorithm is selected to solve the master problem, and Box’s Complex [25] derivative-free optimization algorithm is selected to solve the subproblems. SQP and Complex search engine objects are defined using the template for the search engine class.

The diagram in Fig. 7 illustrates the network configuration for this design problem, depicting data transfer as well as the three workstations to which the objects were bound.

The solution to the problem is

\[
(R^*, L^*, t_b^*, t_f^*) = (41.5, 239.8, 0.801, 5.44)
\]

corresponding to an objective function value of \( 2.27 \times 10^4 \).

In total, 140 calls to the analysis objects were made. Note the ease with which different optimization algorithms can be used if one is not satisfied with the solution, which is often the case in more complicated and perhaps noisy or discontinuous problems.

B. Optimal Design of Integrated Hybrid Powertrain and Engine Simulation

To further demonstrate the functionality of the framework, a hybrid electric vehicle (HEV) powertrain design problem is developed. Simulations of the vehicle (including driveline, electric motor, and energy storage) and the diesel engine are used in the study.

ADVISOR [26] is a feed-forward hybrid electric vehicle simulation. This simulation computes vehicle performance and fuel economy over a driving schedule. The simulation is implemented in MATLAB/SIMULINK, assuming quasi-static states. The vehicle powertrain is configured in parallel, so both an engine and electric motor can provide torque to the wheels by means of a transmission. The engine can also drive the electric motor in reverse as a generator.
to charge a set of batteries. The batteries may in turn power the electric motor when the control strategy deems necessary. Regenerative braking is used, based on a similar principle. Mass and rotational inertia effects of components and vehicle are included in the model. A schematic of a parallel hybrid is shown in Fig. 8. Input variables to the vehicle simulation include motor rated power, battery power and capacity (given in terms of number of battery modules), engine rated power, low and high set points for battery recharge, and transmission gear ratios. The motor rated power, number of battery modules, and engine rated power are used as design variables in this design study.

The engine simulation TDES [27] calculates quasi-static thermodynamic states in the engine cylinder as a function of crank angle. The thermodynamic model combines flow models of a turbocompressor, intercooler, and manifolds with a zero-dimensional, multicylinder diesel reciprocator model. The code is
written in FORTRAN. The output of interest is torque. Input variables to the engine simulation include cylinder dimensions, valve dimensions, compression ratio, operating speed, and fuel mass injected per cycle. TDES is calibrated to simulate a Volkswagen 1.9L TDI engine. The simulation is reversed into a feedback simulation (ADVISOR), to estimate fuel consumed as a function of engine speed and torque. Fuel maps generated by TDES are integrated into ADVISOR at each design iteration. Assanis et al. [28] provide details on engine calibration and integration. Figure 9 shows a diagram of the integrated vehicle and engine models.

Each subsystem model (based on ADVISOR and TDES) is wrapped as a CORBA object, making use of ASCII input and output files. Once wrapped, they can be used in any distributed design effort, without code modification. The TDES engine map generator is spliced into four separate standalone simulations, each creating one-quarter of the engine map.

The design problem is formulated based on the objectives put forward by the Partnership for a New Generation of Vehicles (PNGV). For fuel economy calculation, a combination of US Government urban and highway driving schedules is used. The Federal Test Procedure (FTP) No. 75 driving schedule is equivalent to one stabilized and two transient phases of the Federal Urban Driving Schedule (FUDS). The Federal Highway Driving Schedule (FHDS) is the same as the Highway Fuel Economy Test used by the EPA for Corporate Average Fuel Economy (CAFE) certification. A combined fuel economy metric is defined as follows:
Fig. 9. Hybrid electric vehicle simulation-integration diagram.

\[
\text{Combined} := \frac{1}{0.55 \cdot \text{Urban}_{\text{FTP}}} + \frac{0.45}{\text{Highway}_{\text{FHDS}}} 
\]

In the case of a hybrid electric vehicle, it is necessary to take into account the initial and final states of charge (SOC) of the batteries to determine fuel economy based on charge-neutral runs. This is accounted for by using a technique that iterates on the initial state of charge until the final state of charge is within some tolerance of the initial (in the present case, 0.5%). Constraints in the design problem ensure a charge sustaining vehicle.

The formulation of the optimal HEV design problem is as follows (including simple bounds on component sizes):
Maximize Combined Fuel Economy
with respect to:

- engine (1.0–1.9L), motor (5–25kW), and battery (10–25 modules) sizes
(subject to:
  - \( g_1 \): 0–60 mph time \( \leq 12 \) s
  - \( g_2 \): 40–60 mph passing time \( \leq 5.3 \) s
  - \( g_3 \): 0–85 mph time \( \leq 23.4 \) s
  - \( g_4 \): max. acceleration \( \geq 0.5 \) g
  - \( g_5 \): max. speed \( \geq 85 \) mph
  - \( g_6 \): 5 s distance \( \geq 140 \) ft
  - \( g_7 \): 55 mph cruise grade \( \geq 6.5 \) %
  - \( g_8 \): maximum launch grade \( \geq 35 \) %
  - \( g_9 \): \( \Delta \) SOC for FTP \( \leq 0.5 \) %
  - \( g_{10} \): \( \Delta \) SOC for FHDS \( \leq 0.5 \) %

Besides the simulated Volkswagen 1.9L TDI engine, a Sollectria ACgtx20 AC induction motor and Ovonic NiMH batteries are selected as the main powertrain components, working through a VW five-speed transmission. The vehicle is modeled after a compact car similar to the Opel Astra or the recently introduced parallel-series Toyota Prius.

The user interface component is used to define the problem and to specify how simulations will be executed. The engine map generators run using parallel one-way calls on four separate workstations. The vehicle simulation is called to run after all four maps are complete. The Complex algorithm is used as search engine. A schematic of the problem configuration depicting data transfer, as well as the five workstations to which the objects are bound, is shown in Fig. 10.

The solution to the problem is

\[
\text{(engine size*, motor size*, battery size*)} = (1.4 \text{ L}, 25 \text{ kW}, 18 \text{ modules})
\]
corresponding to an objective function value of 47.4 mpg (combined fuel economy).

Active constraints are the upper bound on the electric motor and the 40–60 mph (passing time) constraint. Performance constraints have the following values at the optimum:

- 0–60 mph time = 10.6 s
- 40–60 mph (passing time) = 5.3 s
- 0–85 mph time = 22.1 s
- max. acceleration = 0.74 g
- max. speed = 91.7 mph
- 5 s distance = 159.5 ft
- 55 mph cruise grade = 6.9%
- maximum launch grade = 38%
A total of 129 calls to the integrated simulation were made. The capability to place engine map generators on distributed workstations significantly decreased the time to obtain the solution.

C. Tracked Vehicle Design for Life Optimization of Road Arm

The design framework is next used for design of the track and torsion bar of an M1-A2 Abrams tank. The configuration for this design problem is shown in Fig. 11. A simulation to estimate load histories on the front most road arm, using a model implemented in DADS, is run on a workstation located at the University of Michigan (UM). This information is used to estimate life expectancy of the road arm, using the Durability and Reliability Analysis Workspace (DRAW) running on a workstation located at the University of Iowa (UI). This setup allows people at different locations to work on separate components of the framework and update and run CORBA servers when new versions of simulations become available.

Design variable values are passed to the load simulator subsystem model DADS. The life-expectancy subsystem model DRAW reads the data from the DADS_RDR server and performs its calculations. The derivative-free algorithm DIRECT [29] is used as the search engine. Further details on this case study are presented by Hulbert et al. [30].
VI. FINAL REMARKS

A distributed object framework is presented for automated large-scale system design. The framework is platform independent and reconfigurable, supports models of various fidelities, accommodates custom and legacy code, makes use of parallel operation, and enables design across distributed networks. Future work on this framework will concentrate on expanding its capabilities by developing more implementations of its components, including search engines, user interfaces, and design coordinators. Several complex case studies are under development to verify functionality and identify potential improvements. These include tracked vehicles, heavy-duty trucks, and passenger vehicles. Demonstrations are performed across distributed networks (e.g., search engines acting on design models at various sites).

ACKNOWLEDGMENTS

This research has been supported by the Automotive Research Center at the University of Michigan, a U.S. Army Center of Excellence in Modeling and Simulation of Ground Vehicles under Contract DAAE07-94-C-R094. This sup-
port and the collaboration of many researchers in the center are gratefully acknowledged.

REFERENCES


18. R. Fellini, Derivative-free and global search optimization algorithms in an object-oriented de-
sign framework, M.S. Thesis, Department of Mechanical Engineering & Applied Mechanics, University of Michigan, Ann Arbor, MI, 1999.


Received April 1999