A Client/Server Framework for On-line Building Code Checking

Charles S. Han[1], John C. Kunz[2], Kincho H. Law[3]

Center for Integrated Facility Engineering
Stanford University
Stanford, CA 94305-4020

Abstract

This paper outlines an integrated client/server framework for an automated code-checking system. Most previous studies have been focused on the processing of design codes for conformance checking. In this work, we examine additional issues: the criteria of a building model, representation of code provisions, the relevance of the provisions with respect to design components, and the encoding of component-based provisions. In this paper, we will demonstrate the integration of these issues in the framework in order to develop an effective system to analyze a design for code-compliance.

Key Words

automation, building codes, handicapped accessibility, product model, STEP, Industry Foundation Classes, client/server, World-Wide Web, Virtual Reality Modeling Language

[1] Graduate Student, Department of Civil and Environmental Engineering, Stanford University, Stanford, CA 94305, csh@galerkin.stanford.edu

[2] Senior Research Associate, Center for Integrated Facility Engineering, Stanford University, Stanford, CA 94305, kunz@cive.stanford.edu

[3] Professor, Department of Civil and Environmental Engineering, Stanford University, Stanford, CA 94305, law@cive.stanford.edu

Introduction

Currently, design and construction documents submitted to a building department for permit-approval are checked manually against a continuously changing and increasingly complex set of building codes. The complexity and the changing nature of building codes leads to delays in both the design and construction processes. The designer must assess which codes are applicable to a given project as well as sort through potential ambiguity in the code provisions. An inspector must go through a similar process. In addition, inconsistencies in interpretation of a given section of the code may differ from inspector to inspector. The design checking and approval process can be a critical activity that prolongs the construction and delays the operation of a facility. Automating this process has the potential to alleviate both the delays and inconsistencies associated with manual checking by giving the designer and the permit-issuing body a consistent framework in which to apply and check codes.

Several researchers have developed frameworks for the representation and processing of design standards (de Waard92) (Kiliccote96) (Yabuki92). A survey of developments for computer representation of design codes was reported by Fenves, et.al. (Fenves94). In this study, we focus on the problem to develop a framework for architectural building code issues by initially investigating handicapped accessibility. Among the numerous provisions governing a facility design, the two issues that have been identified by facility managers as most significant are accessibility and egress. Though we are examining a self-contained aspect of the building code, there is sufficient complexity and ambiguity in the handicapped accessibility code that warrants a close examination of the issues that are fundamental to the development of a practical integrated framework for the representation and processing of design standards.

The design intent of the handicapped accessibility code is to provide the same or equivalent access to a building and its facilities for disabled persons (for example, persons restricted to a wheelchair, persons with hearing and sight disabilities) and persons without qualifying disabilities. To fulfill this intent, organizations such as the U.S. Access Board (also known as the Architectural and Transportation Barriers

Compliance Board), the authors of the Americans with Disabilities Act (ADA), have developed prescriptive measures such as various clearances and reach thresholds for building components. For example, the U.S. Access Board has developed minimum clearances to allow transfer of a person from a wheelchair to a toilet and minimum lengths of grab bars associated with a toilet. In the example presented in this paper, we have only implemented issues related to door accessibility, but the concepts of clearance and distance thresholds are similar for other building components that must be accessible. Therefore, the developed prototype can potentially be extended to accommodate the critical building components that must be accessible.

Advances in Internet and web-based technologies will have a significant role in making on-line code-checking a reality (Han97). Specifically, on-line checking of building designs via the World Wide Web (WWW) can be organized in a client-server environment. The development of a standard product model including the Standard for the Exchange of Product Data (STEP) (ISO94) and the International Alliance of Interoperability (IAI) Industry Foundation Classes (IFC) (IAI97) will further facilitate design data exchange. In this work, the user (the client) develops a plan using an IFC-compliant CAD package (enhancements have been made to AutoCAD to output an IFC EXRESS file from a building design). At any point in the design process, the user can send this design to a code-checking program that resides on a remote server. The code-checking program examines the IFC design data and summarizes the results in a generated web page. The web page contains a graphical representation of the building model along with "redline" information with hyperlinks to specific comments, and, when applicable, the comments have hyperlinks to the actual building code document provisions (in this case the Americans with Disabilities Act Accessibility Guidelines (ADAAG) (ADAAG97)).

We examine the structure and attributes of a product model and a building code model needed to provide sufficient design information to be analyzed by a code-checking program. The code-checking program must be able to read the design data and reorganize the information in a form that can be analyzed against a model of the building code. We describe this building code model as a mapping of building code provisions to methods that are embedded in an object-oriented framework appropriate to analyze a design for code-compliance. We categorize the building code into three classes:

1. provisions that determine the relevancy of other provisions
2. provisions that examine the criteria of a system of building components
3. provisions that examine the criteria of individual building components.

This paper examines the integration of a building product model and the object-oriented building code model.

The Building Product Model

In order to automate the checking of a building design for compliance to a building code document, the program that does the analysis of a building design must understand the design data. Currently, drawings that are manually inspected by a building department are two-dimensional representations of three-dimensional information, a constraint of using paper as the medium of communication. An inspector must coordinate related drawings such as plans with elevations in order to develop a three-dimensional image to check a design against a building code. Three-dimensional model development and analysis has several advantages over viewing and interpreting a two-dimensional representation:

- Representation of building components and their geometrical relationships to other components is explicit (there is no ambiguity as there is with the interpretation and construction of the three-dimensional building model from two-dimensional representations which often have missing or contradictory data)
- Elimination of the need (albeit physical models or mental images) of the three-dimensional building model from two-dimensional views

- Function and behavior of the building model components can be more accurately modeled in a three-dimensional representation.

Some current CAD systems allow the designer to develop three-dimensional building models, but when designers assemble a design or construction document package, they typically generate two-dimensional representations (plans, elevations, sections) for the review process. Inherent in the projection of the three dimensional representation to a set of two-dimensional representations is the loss of design data, the simplest of which is the loss of data of one of the three dimensions. Directly using three-dimensional building models for analysis purposes alleviates the need for the process of projection and subsequent regeneration (2D back to 3D).

There have been several research efforts to develop object-oriented CAD systems and object-oriented building models that contain the necessary geometric, functional, and behavioral relationships of building components (de Waard92) (Garrett89) (Ito89). Currently, there is an effort by the International Alliance of Interoperability (IAI), a consortium of CAD vendors and other AEC industry partners, to develop standards for a three-dimensional project model that enables interoperability between applications by different software vendors (IAI97). The IAI's effort includes defining a set of objects called Industry Foundation Classes (IFCs) that conform to current object-oriented philosophy. IFC Release 1.0 currently defines two standard formats for sharing project data: a standard EXPRESS file format and software interfaces. The development of IFC-compliant CAD and analysis packages to enable interoperability is the main goal of the IAI. We have chosen to use the IFC Release 1.0 project model as our point of departure for the building model.[4] For our prototype, we implemented the IFC hierarchy as shown in Figure 1.

In this work, AutoCAD is being employed as the design environment. We have developed a simple AutoCAD-to-IFC translation module generating the building model

[4] IFC Release 1.5 was made available in early 1998, and Release 2.0 is being planned to become available in early 1999. Entity classes differ between Release 1.0 and Release 1.5, but the Release 1.0 model adequately describes a building design for the code-checking prototype described in this paper.

class attributes and relationships required by the code-checking program to perform the building code analysis. Since AutoCAD currently does not support the IFC building model format, we have created an additional layer of building component objects with semantics corresponding to the IFC specifications (see Figure 2 for an example of IFC-specific relationships of building components). At any point in the design process, the designer can send the building model to the code-checking program that resides on a remote server. An AutoLisp routine extracts the IFC information from the enhanced AutoCAD database and converts the information into an IFC EXPRESS file. An example of a building model and the corresponding IFC EXPRESS file is shown in Figure 3.[5]

For the code-checking program to correctly analyze the building model, we make one assumption about the use of the IFC `IfcSpace` that describes the attributes of a space, that is the designer needs to make explicit whether a space needs to be accessible. We examine this issue in detail in the section that describes relevancy as it is implemented in the building code model.

A Client/Server Framework

Our prototype is built utilizing a client/server environment as shown in Figure 4. The code-checking program resides on the server side of the system. Both the client and the server are written in Java giving both sides of the system platform independence. The user (the client) sends an IFC model (an IFC EXPRESS file) across the network to the code-checking program. The abstraction of network operations in Java made constructing a proprietary client/server communication sequence a relatively easy task. The code-checking program is continually listening to a predetermined socket on the server for the appropriate start sequence and the subsequent stream of IFC EXPRESS data. In our prototype, an AutoLisp routine spawns the client process after it has generated the IFC

[5] This example is shown to illustrate the IFC information generated by a simple building design; the EXPRESS file generated by a building design, for instance, in the code-checking example is too lengthy to be included in a figure.

EXPRESS file. Note that the client program can also be started independently of the AutoLisp routine and will execute successfully as long as there is a generated IFC EXPRESS file that is ready to be sent to the code-checking program. The client opens the IFC EXPRESS file and the corresponding socket on the server, sends the appropriate start sequence, and then sends the stream of the IFC EXPRESS file to the code-checking program. The results are posted to a web page, and the code-checking program notifies the client that the analysis is complete.

The code-checking program stores the IFC data as objects in a data structure that uses objects based on the hierarchical structure of the IFCs as shown in Figure 1. We have constructed our Java IFC classes directly mapping the attributes and relationships as defined in the IFC EXPRESS schema thereby retaining the desired behavior and functionality (for example comparisons, see Figure 5 and Figure 6). The IFC objects in the data structure have slots to accommodate additional information that is generated by the code-checking program (such as additional graphical information and comments associated with either code-compliance or code violations).

The code-checking program on the server reads the IFC data sequentially, and an object that makes reference to a previously instantiated object will simply point to the previous object in the appropriate attribute field. Certain attributes of a building component are described by a subsequent building component instantiated on another line in the EXPRESS file. For example, when an `IfcDoor` is instantiated, an `IfcRelFills` and an `IfcRelVoids` must be instantiated in order to make the association between this door and the appropriate opening (see the EXPRESS file in Figure 3, specifically lines **#34**, **#60**, **#61**, and **#62**). Once all the IFC data has been put into a data structure containing the building components, the code-checking program is ready to analyze the building model.

The Building Code Model

The building code model uses the same structure as the IFC project model hierarchy. For example, all encoded provisions concerning door accessibility would be instances of a door accessibility class. Therefore, an individual component can be checked against all the applicable instances of the provisions for that class of building component. Since IFCs are grouped by similar functional units (for example, doors and windows are subclasses of a class called `IfcFillingElement`), by design, we can group similar code issues. For example, doors and windows have egress-related provisions. By structuring the encoded provisions in this manner, we loosely categorize building component provisions by design intent since the behavior of similar building components is similar. Design intent is defined differently according to the context (Garza95). Here, we are specifically examining the design intent of the code provision.

The code-checking program reads in a stream of IFC data to populate its database of building components. Similarly, the program reads in a stream from a building code file. This file is a mapping from the text of provisions of a building code document to an EXPRESS file that has instances of the encoded provisions. The code-checking program reads in the building code EXPRESS file and populates a data structure containing instances of the building code provisions. In the current implementation, since we are focusing on the ADAAG, an ADAAG building code file containing encoded provisions is read once when code-checking program is initialized on the server. Figure 7 shows an example of the EXPRESS schema for a handicapped-accessibility building code class, the corresponding Java class, and sample lines from the ADAAG EXPRESS file that are instances of the handicapped-accessibility building code class.

Notice that one building code class can have several instances that correspond to related provisions in the building code—in this case, there is a class of building code corresponding to the issue of door clearance. Here, a handicapped-accessibility building code class is an abstraction of a handicapped-accessibility concept, and the instances in the ADAAG EXPRESS file map to the actual text of the building code provisions, in this case the ADAAG document. Other handicapped accessibility building codes can be easily mapped to this accessibility code class hierarchy and generate corresponding EXPRESS files.

Relevance

Relevance as applied to checking a building design against a building code is applied in the following ways:

- Determining which provision or provisions are applicable to a given building component or system of building components.
- Conversely, determining which building component or system of building components are applicable to a given provision or set of provisions.
- Resolving exceptions within a provision.

There are several levels of relevance that need to be addressed. We take a top down approach to determine the relevance of provisions for a specific building component. First, we decide if a set of provisions is relevant to the project under consideration (the project model). Next we look at specific buildings, specific floors, and then specific spaces (for example, rooms). Note that this hierarchy follows the same class hierarchy as the IFC model—it is logical that the structure of the code model closely follows the structure of the IFC class hierarchy. Finally, we determine if a set of provisions is relevant to the specific building component associated with a specific space.

We can determine whether provisions are relevant to a project or building given the same information that building inspectors receive in construction documents. For example, a set of plans must list the type of building (for example, commercial, residential, and so on). However, our relevance engine must determine whether a set of provisions is applicable to a given space in the building product model. There are instances when exclusions need to be made explicit such as delineation of a historical space, but this additional information must be made available to the building inspector who is checking the plans.

Currently, in our implementation of the relevance module, we are looking only at relevance issues related to accessibility on the space level. The ADAAG sets guidelines for determining if a door is accessible, but other provisions must be analyzed to determine whether the door accessibility provisions are applicable to a specific door in the building model. We require the user to explicitly label a space (an `IfcSpace`) as accessible (in our implementation, if the name of a space starts with the letter '`x`,' then accessibility is not required in this space) and the code-checking program examines this information to note which spaces are accessible. There are several states that a building component can have. The code-checking program initially labels a space as either `REQUIRED` or `NOT_REQUIRED` (for accessibility) according to how the user has explicitly labeled the space. The other state that co-exists with the `REQUIRED` state, `PASSING` or `FAILED`, will be examined in the next section.

The code-checking program then determines which building components are associated with a specific space that is in the `REQUIRED` state. If a building component is within or intersects the space, it is put into the set. One of the attributes for `IfcSpace` is the container `HasElements`, and we use it for this purpose. We could generate this information in the CAD environment; however, since we cannot predict how individual IFC implementers will use this container, we let the code-checking program generate this information.

The code-checking program is now ready to analyze the building components in each space that has been labeled `REQUIRED`. After the building components are analyzed, the code-checker must again determine whether an individual building component that is contained in a `REQUIRED` space is relevant for the overall code-compliance of the building design. This is the issue of cardinality deciding whether a building component needs to be accessible in relation to the other building components in the space being examined. Since higher-level provisions often determine the number of similar building components that need to comply to a specific building code issue, an individual building component must be examined in relation to other building components of the same class within a given space. For example, not all water closets in a space need to be accessible.

Therefore, the code-checking program must first analyze the building components in a space and then group and re-analyze the subset of similar components to determine whether they finally need to comply with accessibility code provisions. If a building component does not meet accessibility code-compliance, it may not mean that the project is in violation of the accessibility code. For example, the building code may only require that there exists a similar building component in the same space that complies.

Encoded Building Component Provisions

The code-checking program analyzes the building model twice with respect to the building code instances. First, it analyzes all the building components in the `REQUIRED` spaces. Then it determines whether a component needs to comply with the building code provisions relative to the other components in the space.

We have concentrated on encoding building component provisions that can be mapped to methods in our building code model. Geometric tests are examples of easily encodable building-component-based provisions (for example, the clearance requirements of a door for egress or accessibility). Encoding these provisions alone is not sufficient for the building code framework—the code-checking program must apply relevance tests to the specific building component or components in question to see whether a provision or set of provisions is applicable.

We noted that the code-checking program generates sets of building components associated with particular spaces that need to be checked for accessibility. Conversely, each building component is associated with a set of one or more spaces (for example, building components such as doors may be associated with two spaces since a door can connect two spaces). Once the code-checking program establishes these relationships, it examines all the building components including components that are associated with a space that are not required to be accessible (it is useful for the designer to have as much information about all the building components in a design).

A building component can have several states. We have already established the accessibility state of spaces (REQUIRED or NOT_REQUIRED), but the code-checking program cannot yet determine if an individual building component associated with a REQUIRED space necessarily needs to be accessible; compliance of one component may be dependent on one or a set of other components. Each component is checked against the relevant provisions, and after all related components are checked, the code-checker determines whether the set of components is in compliance. The building component initially has a state where the requirement for accessibility is unknown and has the state of PASSING until it fails one of the encoded provisions tests.

The code-checking program examines each of encoded provisions in the building code instances until it matches two attributes: the class of the building component and the attribute IfcClass in the building code component and the attribute level with the string "ELEMENT." In this inspection of the building model, the code-checking program is examining building components on the element level as opposed to the space level. When the code-checking program finds a match, it tests the encoded provision (the building code component instance) against the building component. Most of the encoded provisions are geometric tests. For those that are associated with the issue of clearance, the building component must be checked against other building components within the associated accessible space (or in the case of a door, the two accessible spaces that it connects).

If the building component complies with the encoded provision, the code component returns a PASSING status. If not, it returns a FAILED status (here, FAILED sets a status bit to TRUE) along with text comments. The return status is ORed with the current status and the module continues to traverse and search for relevant encoded provisions. The building component's status is ORed because if a building component has failed a previous test, passing a future provision does not change its non-compliant state. The code-checking program continues to check a component even if it has failed a previous test so it can return as much information on a building component as possible. For example, a door can fail several clearance tests, and all issues may be of interest to the designer. As noted earlier, we reserve some fields in a building component for issues outside the scope of the IFC. The code-checking program uses these fields to append the comments associated with code violations.

After examining all of the building components the code-checking program must resolve whether an individual building component within an accessible space is required to comply with the accessibility code. The code-checking program makes a second pass through the database of building components this time looking for matches with code component attribute `IfcClass` and the type of building component. The program also looks for matches with the code component attribute "level" with the string "`SPACE`." The code-checking program concurrently examines a set of identical building components associated with the same space (or spaces as would be the case for doors). For example, an ADAAG provision states that if toilet stalls are provided, then at least one needs to be accessible (this provision has several more qualifying statements, but using the simplified provision demonstrates the reasoning method used to analyze the particular space). The code-checking program must examine each toilet associated with the accessible space. If there is one that complies with the accessibility code (it had previously been marked as `PASSING`), the code-checking program marks this toilet as `REQUIRED` and all the others as `NOT_REQUIRED`. If there are no toilets in the accessible space (they had previously been marked as `FAILED`) that comply with the accessibility code, then the code-checking program marks all the toilets in the space as `REQUIRED`. This serves as a notification to the designer that none of the toilets in the space meet the accessibility code, and at least one needs to be compliant.

At this point, the code-checking program has finished the analysis of the building model. All building code issues (compliance or non-compliance, associated comments) have been attached to the building components. The final step is to generate the information in the form of a web page.

Web Page Generation

The code-checking program examines building component database one final time and extracts the necessary information to generate a web page consisting of three frames: a VRML frame, `COMMENTS` frame, and a building code document frame. In our prototype, the code-checking program color-codes a building component in the VRML model according to the rules in Table 1.

Color-coding a building component yellow is useful for a designer to realize that even though the specific component need not be accessible, it does not comply. In addition to the color-coded information, all VRML objects are hyperlinked to a set of comments in the `COMMENTS` frame since it is useful to know that an object complies with the building code. As the VRML components are generated, since they are linked to the `COMMENTS` frame, the `COMMENTS` frame is concurrently and dynamically generated with the corresponding HTML anchors. The links in the `COMMENTS` frame to the building code document frame have been predetermined for each encoded provision (see examples of encoded-provisions instances in Figure 7). The building code document (in this case, the ADAAG) already has the predefined anchors associated with the possible links that are generated in the `COMMENTS` frame.

Example

To illustrate the framework developed, we present a simple example of a building model that does not comply with the ADAAG. We step through the reasoning of the code-checking program and present the results. Then, we modify the example to make the design comply and highlight the differences in the code-checking analysis. For this example, we focus on door accessibility issues. We will also underline the advantages of using a three-dimensional building model by providing an analogous analysis of a two-dimensional representation of the building model.

A Non-Compliant Design

Figure 8 shows a plan view and an isometric view of the example. All spaces except for the one noted have been marked by the designer as spaces where accessibility is required. Doors **`D1`** and **`D2`** violate ADAAG door width provisions, and the half-wall interferes with one of the maneuvering clearances for door **`D3`**. All other doors in the design meet accessibility requirements so we will focus the discussion on these three doors.

Relevance

The code-checking program marks door **`D1`** as `NOT_REQUIRED`. **`D1`** does not need to comply with accessibility provisions since it serves a space (space **`X`**) that is not required to be accessible. For example, space **`X`** might be a mechanical room. Since all other spaces are required to be accessible, the code-checking program cannot yet determine if the other doors in the space need to comply with the accessibility provisions. The code-checking framework now checks each door in the building model.

Encoded Provisions

The code-checking program checks all the doors against the relevant provisions that examine individual building components in relation to the whole building model. Doors **`D1`** and **`D2`** violate width requirements and **`D3`** violates a maneuvering clearance provision. At this point, the code-checking program contains the status information in Table 2.

Relevance Revisited

Finally, the code-checking program checks the doors against the relevant provisions that relate to the cardinality issue. For doors, an accessible space must have at least one door on an accessible path (a simplified view, but it demonstrates the issue of cardinality). **`D1`** has already been taken care of since accessibility is not required. An accessible route is required between space **`A`** and the corridor to its left. The two door candidates for an accessible route are **`D2`** and **`D3`**. Since both violate accessibility code provisions and there is no accessible route between the two spaces, the code-checking program labels both doors as being required to meet accessibility provisions shown in Table 3.

Generating the Web Page

Figure 9 shows the web page generated for the non-compliant building design. Doors **`D1`**, **`D2`**, and **`D3`** in the VRML model are hyperlinked to comments **`#45`**, **`#46`**, and **`#49`** respectively. Door **`D1`** (not shown) is color-coded yellow, and **`D2`** and **`D3`** are color-coded red. The underlined portions of the comments are hyperlinked to the appropriate ADAAG provisions. The code-checking program also generates semi-transparent redline clearance box associated with **`D3`** and the appropriate encoded maneuvering clearance provision. It is hyperlinked to the appropriate sub-comment in comment **`#46`**.

Two-Dimensional Representation Analysis

Figure 10 shows the necessary two-dimensional representations needed to carry out the analogous analysis of the same example. A code-checking program analogous to our prototype would first have to reconstruct the building model from the appropriate 2D representations in order to check it for code compliance. We assume that whatever modeling package is employed to generate the 2D representations, the 2D model will have a logical naming scheme for the building components. For example, **`A-D1`**, **`B-D1`**, and **`C-D1`** would correspond to the same door as viewed in the plan and the two elevations. Even with some standard naming convention, there is no guarantee of completeness of the generated 3D model. Without a naming convention, the analogous 2D code-checking program would have to utilize algorithms to match different views of the same component.

The building model in Figure 10 illustrates some additional problems associated with this regeneration task. The number of 2D views needed to generate sufficient information depends on the building model. Here, we must generate two elevations in order to get all the height information of the half-wall and door **`D1`**. More views would be needed for non-orthogonal components and placement of components.

As with the 3D-model analysis, the analogous 2D code-checking program can determine that doors **`D1`** and **`D2`** violate the width requirement information from any of the views. In this case, each 2D view is sufficient. Note, though, that if door **`D2`** were widened to meet the width requirements, Figure 10-A would not be adequate to guarantee compliance since it gives no height information, and Figure 10-B or Figure 10-C would be needed to generate the 3D model or provide the additional height information. In the case of door **`D3`**, any individual 2D view in Figure 10 is insufficient to determine whether the half-wall interferes with the maneuvering clearance, and here the code-checking program must generate the 3D information from Figure 10-A and Figure 10-B.

The Revised Design

When we remove the half-wall, the design now complies with the ADAAG. The code-checking program's analysis is identical to the previous example except that **`D3`** is given a `PASSING` status. Since **`D3`** provides an accessible route between space **`A`** and the corridor to its left, **`D2`** is no longer required to meet the accessibility provisions. Figure 11 shows the web page generated for the revised and compliant design. Again, all the accessibility information is generated for each building component. In the associated VRML frame, **`D2`** is color-coded yellow, and **`D3`** is rendered showing it meets the accessibility code requirements. The information generated by the code-checking program when it examines the doors against the relevant space-level provisions is shown in Table 4.

Discussion

In this paper, we have demonstrated the modules of an integrated framework for code checking with specific focus on accessibility issues. Although only provisions related to door accessibility have been implemented, the key modules required to deliver the client/server code-checking framework have been formalized and completed. The prototype is easily extendible to accommodate other building components that must be accessible. While the code-checking program is successful in handling encodable provisions and localized door compliance, it does not address the following issues:

- Complete handling of the relevance issue
- Ambiguous provisions
- Complete system-wide accessibility analysis

In the following, we outline future research that is necessary to address these shortcomings in the framework.

Framework Enhancements

Figure 12 outlines our proposed code-checking framework delineating the future enhancements that are discussed in this section.

Relevance

We must increase the capabilities of the building code framework so the code-checker can determine which spaces must be made accessible in the building design. Currently, we must explicitly define the spaces that are accessible whereas it is desirable for the code-checker to make the determination. There are certain gray areas—in certain circumstances, a space may be explicitly delineated as not needing to be accessible. For example, in a historically significant building, current provisions may not be relevant to some of the spaces. Therefore, we must formally define the desired space-analyzing capabilities of the code-checker.

We must also increase the scope of the relevance module by taking into consideration the design intent behind certain provisions. For example, when we expand our code-checking program to handle egress, we must address the following relevance problem: Building codes explicitly state when exit signs are required based on occupancy type. Here, the design intent is based on the issue of familiarity. A person does not need exit signs in one's home because a resident is probably familiar with the possible exit paths whereas in buildings with which occupants may be unfamiliar with the surroundings, these paths require signs. Also, if the room is small enough (for example, a small office), even if the occupant may be unfamiliar with the space, the exit path is more obvious and a sign is not required. Formally, when the search space is below some threshold (here, familiarity reduces the search space), exit signs are not required.

Brokered Lookup and Integration of the Distributed Object Environment

Certain building code provisions are purposely left ambiguous. In following example from the ADAAG, the U.S. Access Board has not specifically defined what constitutes *a shape that is easy to grasp*:

> **4.13.9 Door Hardware.** Handles, pulls, latches, locks, and other operating devices on accessible doors shall have a shape that is easy to grasp with one hand…

In order to resolve such provisions, we propose to defer the decision as to whether an individual building component complies with the building code to another source that keeps a repository of compliant building components for a specific provision. We propose to implement a distributed object environment such as CORBA to implement this brokered lookup in our framework. However, if we are going to implement this environment, it makes sense to implement this technology for an overall shared product model environment.

Currently, we envision a shared product model as a repository of building components on the network (see Figure 13). Various disciplines will have different views and different capabilities of modifying the project model. For example, a CAD view might be graphical and a cost analysis view might be in the form of a spreadsheet. Code-checking would be one of the many services (various different areas of code-checking would in fact be different services) that are on the network. These services would be invoked from a view—for example, code-checking might be invoked from the CAD view, a search for best prices on materials might be invoked from the cost analysis view. The invocation would go through a trading service that would make the determination of the relevant service. The trading service would then initiate the transfer of the shared project model (or relevant parts of the model) to the relevant service. In this environment, the brokered lookup would simply be a sub-service of the code-checking framework. The details on information transfer need to be formalized, but the current state of distributed object technology allows such shared product model scenarios to be implemented.

Simulation

We are currently working on the implementation of the motion planning algorithms to address global accessibility issues through simulation. Static encoded provisions that test local clearance issues are insufficient to extract accessible path information from an entire building design. In general, simulation has advantages over static encoded provisions to examine system-level issues while static encoded provisions may be sufficient to analyze local phenomena or individual building components.

Conclusion

We seek to solve the design standards processing problem in one domain (in this case, handicapped accessibility) and then apply the principles of our code-checking framework to other domains. Since our approach entails certain domain-specific solutions, we do not present a completely generalized solution. However, the principles behind our approach to building code checking developed here for disabled access should be applicable to other aspects of the building code.

Exploration of the amount of information contained in a building model is an important issue. Attributes related to a building code should be generated by the code-checking program as opposed to being explicitly defined within the building model. As an example of explicitly defined attributes, IFC `IfcSpace` has among its attributes `ExitPaths`, `ExitWidths`, and `ExitDistances` (IAI97). Subsequent releases of IFC will define more code-related processes and attributes such as `IfcRamp` to explicitly define a ramp building component. If a designer is going to employ the services of a code-checking program, the program, not the designer, should make the determination of whether an exit path exists. In addition, when the building code changes, the integrated building/building code model will be outdated. Our current prototype requires that the designer explicitly state whether a space requires accessible access—this requirement will be eliminated as we further develop the building code framework.

We have demonstrated the feasibility of online code-checking with our framework. We continue to develop the framework and implement all the proposed enhancements to fully realize the significance of our approach to the code-checking problem. In our approach, we seek to solve the specific problem of automating the disabled access analysis of a building design. This paper has outlined the steps needed to solve a specific code-checking problem, and we hope to extend our approach to make it applicable to other areas of code-checking, but by no means do we feel that we are solving all the generalized design standards processing issues.

Acknowledgments

This research is partially sponsored by the Center for Integrated Facility Engineering at Stanford University. Discussions with facility managers have given valuable insight into the building code review process.

References

(ADAAG97) Access Board (U.S. Architectural and Transportation Barriers Compliance Board) (1997). *Americans with Disabilities Act Accessibility Guide*, Washington, DC.

(de Waard92) de Waard, Marcel (1992). Ph.D. Thesis: Computer Aided Conformance Checking: Checking Residential Building Designs Against Building Regulations with the Aid of Computers, The Hague, The Netherlands.

(Fenves94) Fenves, S.J., Garrett, J.H., Kiliccote, H., Law, K.H., Reed, K.A. (1995) “Computer Representations of Design Standards and Building Codes: U.S. Perspective,” *The International Journal of Construction Information Technology*, University of Salford, Salford, U.K.

(Garrett89) Garrett, J.H., Basten, J., Breslin, J., Andersen, T. (1989) "An object-oriented model for building design and construction," *Proc. Struct.Congress,* ASCE pp. 332-341, New York, NY.

(Han97) Han, C.S., Kunz, J.C., Law, K.H. (1997) "Making Automated Building Code Checking a Reality," *Facility Management Journal,* IFMA September/October 1997 pp. 22-28, Houston, TX.

(IAI97) International Alliance for Interoperability (1997). *Industry Foundation Classes Release 1.0, Specifications Volumes 1-4*, Washington DC.

(ISO94) International Standards Organization, Technical Committee 184, subcommittee 4 (1994). *ISO 10303-1:1994: Industrial automation systems and integration—Product data representation and exchange—Part 1: Overview and fundamental principles,* Geneva, Switzerland.

(Ito89) Ito, K., Ueno, Y., Levitt, R.E., Darwiche, A. (1989) "Linking knowledge-based systems to CAD design data with an object-oriented building product model," *Technical Report 17, Center for Integrated Facility Engineering*, Stanford University, Stanford, CA.

(Kiliccote96) Kiliccote, Han (1996). Ph.D. Thesis: *A Standards Processing Framework*, Carnegie Mellon University.

(Vogel97) Vogel Andreas, and Keith Duddy (1997). *Java Programming with CORBA*, John Wiley and Sons, Inc., New York, NY.

(Yabuki92) Yabuki, N. and Law, K.H. (1992). "An Integrated Framework for Design Standards Processing," *Technical Report 67, Center for Integrated Facility Engineering*, Stanford University, Stanford, CA.

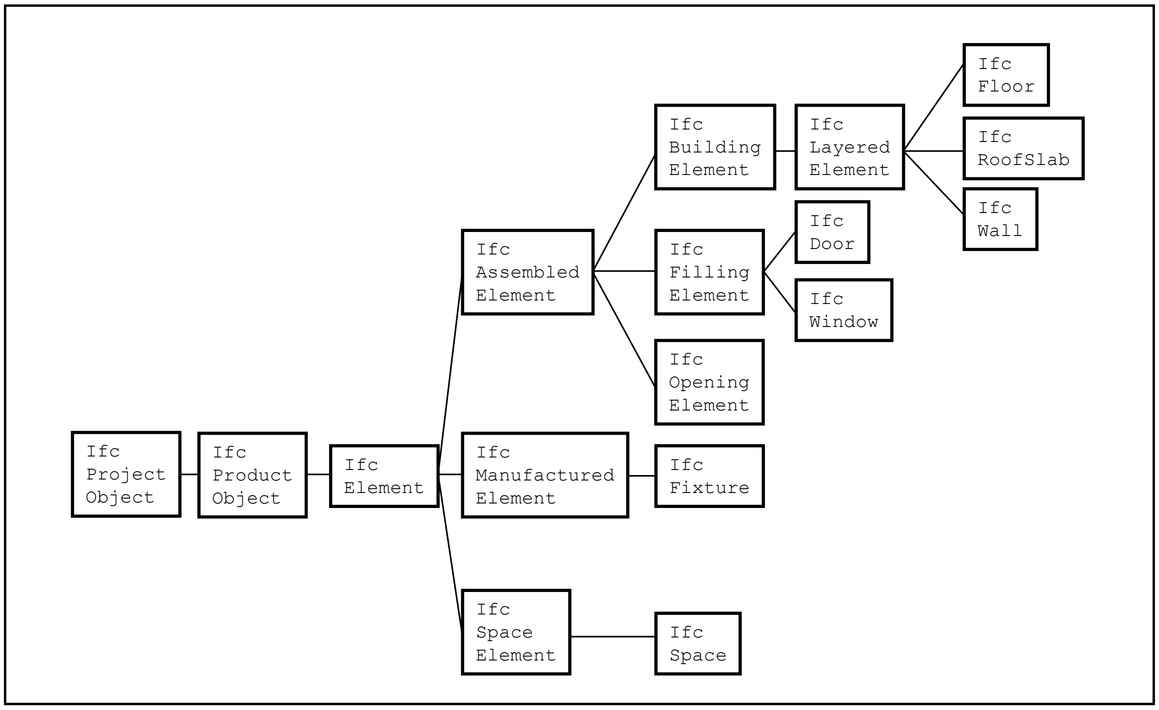
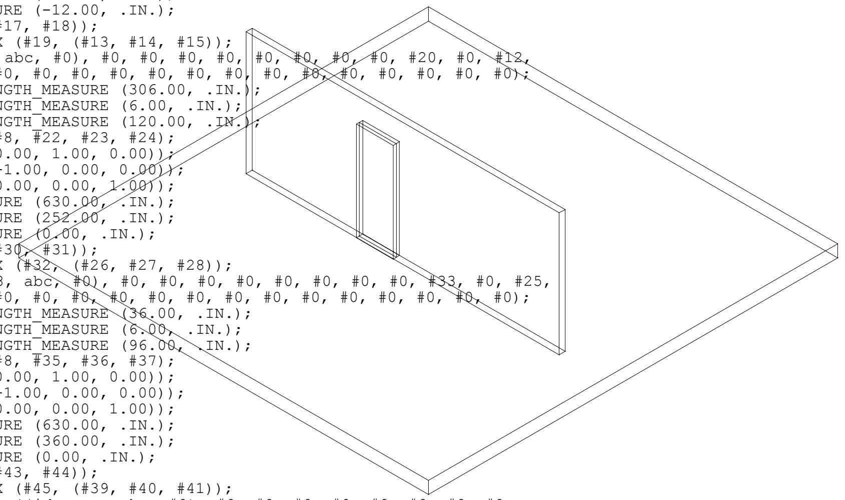


Figure 1: IFC class hierarchy implemented.

Figure 2: An example of a relationship between door and wall classes.



```
#1 = P42_DIRECTION ((1.00, 0.00, 0.00));
#2 = P42_DIRECTION ((0.00, 1.00, 0.00));
#3 = P42_DIRECTION ((0.00, 0.00, 1.00));
#4 = P41D_LENGTH_MEASURE (0.00, .IN.);
#5 = P41D_LENGTH_MEASURE (0.00, .IN.);
#6 = P41D_LENGTH_MEASURE (0.00, .IN.);
#7 = IFCPOINT ((#4, #5, #6));
#8 = IFCORIENTEDVERTEX (#7, (#1, #2, #3));
#9 = P41D_POSITIVE_LENGTH_MEASURE (400.00, .IN.);
#10 = P41D_POSITIVE_LENGTH_MEASURE (400.00, .IN.);
#11 = P41D_POSITIVE_LENGTH_MEASURE (12.00, .IN.);
#12 = IFCBOUNDINGBOX (#8, #9, #10, #11);
#13 = P42_DIRECTION ((1.00, 0.00, 0.00));
#14 = P42_DIRECTION ((0.00, 1.00, 0.00));
#15 = P42_DIRECTION ((0.00, 0.00, 1.00));
#16 = P41D_LENGTH_MEASURE (440.00, .IN.);
#17 = P41D_LENGTH_MEASURE (200.00, .IN.);
#18 = P41D_LENGTH_MEASURE (-12.00, .IN.);
#19 = IFCPOINT ((#16, #17, #18));
#20 = IFCORIENTEDVERTEX (#19, (#13, #14, #15));
#21 = IFCFLOOR ((site, abc, #0), #0, #0, #0, #0, #0, #0, #0, #0, #20, #0, #12,
      #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0);
#22 = P41D_POSITIVE_LENGTH_MEASURE (306.00, .IN.);
#23 = P41D_POSITIVE_LENGTH_MEASURE (6.00, .IN.);
#24 = P41D_POSITIVE_LENGTH_MEASURE (120.00, .IN.);
#25 = IFCBOUNDINGBOX (#8, #22, #23, #24);
#26 = P42_DIRECTION ((0.00, 1.00, 0.00));
#27 = P42_DIRECTION ((-1.00, 0.00, 0.00));
#28 = P42_DIRECTION ((0.00, 0.00, 1.00));
#29 = P41D_LENGTH_MEASURE (630.00, .IN.);
#30 = P41D_LENGTH_MEASURE (252.00, .IN.);
#31 = P41D_LENGTH_MEASURE (0.00, .IN.);
#32 = IFCPOINT ((#29, #30, #31));
#33 = IFCORIENTEDVERTEX (#32, (#26, #27, #28));
#34 = IFCWALL ((iwall-8, abc, #0), #0, #0, #0, #0, #0, #0, #0, #0, #33, #0, #25,
      #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0);
#35 = P41D_POSITIVE_LENGTH_MEASURE (36.00, .IN.);
#36 = P41D_POSITIVE_LENGTH_MEASURE (6.00, .IN.);
#37 = P41D_POSITIVE_LENGTH_MEASURE (96.00, .IN.);
#38 = IFCBOUNDINGBOX (#8, #35, #36, #37);
#39 = P42_DIRECTION ((0.00, 1.00, 0.00));
#40 = P42_DIRECTION ((-1.00, 0.00, 0.00));
#41 = P42_DIRECTION ((0.00, 0.00, 1.00));
#42 = P41D_LENGTH_MEASURE (630.00, .IN.);
#43 = P41D_LENGTH_MEASURE (360.00, .IN.);
#44 = P41D_LENGTH_MEASURE (0.00, .IN.);
#45 = IFCPOINT ((#42, #43, #44));
#46 = IFCORIENTEDVERTEX (#45, (#39, #40, #41));
#47 = IFCOPENINGELEMENT ((idoor-c, abc, #0), #0, #0, #0, #0, #0, #0, #0, #0,
      #46, #0, #38, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0);
#48 = P41D_POSITIVE_LENGTH_MEASURE (36.00, .IN.);
#49 = P41D_POSITIVE_LENGTH_MEASURE (2.00, .IN.);
#50 = P41D_POSITIVE_LENGTH_MEASURE (96.00, .IN.);
#51 = IFCBOUNDINGBOX (#8, #48, #49, #50);
#52 = P42_DIRECTION ((0.00, 1.00, 0.00));
#53 = P42_DIRECTION ((-1.00, 0.00, 0.00));
#54 = P42_DIRECTION ((0.00, 0.00, 1.00));
#55 = P41D_LENGTH_MEASURE (628.00, .IN.);
#56 = P41D_LENGTH_MEASURE (360.00, .IN.);
#57 = P41D_LENGTH_MEASURE (0.00, .IN.);
#58 = IFCPOINT ((#55, #56, #57));
#59 = IFCORIENTEDVERTEX (#58, (#52, #53, #54));
#60 = IFCDOOR ((idoor-c, abc, #0), #0, #0, #0, #0, #0, #0, #0, #0, #59, #0, #51,
      #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0, #0);
#61 = IFCRELFILLS (#47, (#60));
#62 = IFCRELVOIDS (#34, (#47), ('foo', 'TRUE'));
```

Figure 3: Graphical representation of a simple design and the corresponding IFC EXPRESS file.

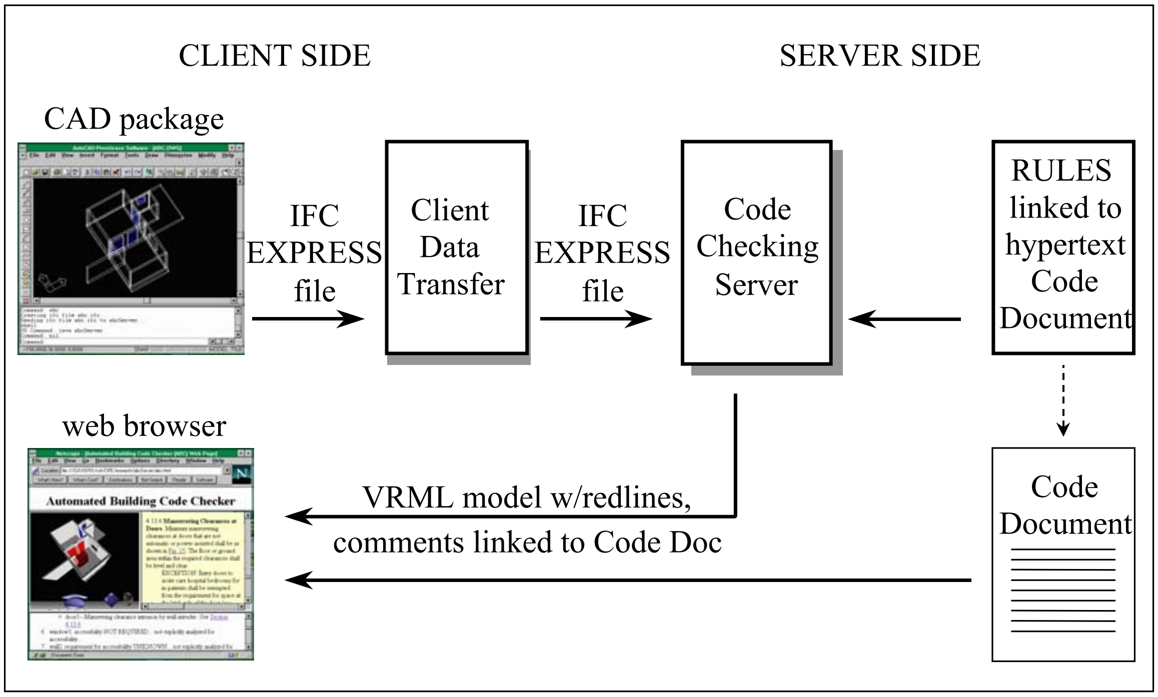


Figure 4: Mechanics of the prototype code-checking program.

IFC EXPRESS schema for IfcFillingElement:

```
ENTITY IfcFillingElement
    ABSTRACT SUPERTYPE OF (ONEOF
        (IfcDoor,
         IfcWindow))
    SUBTYPE OF (IfcAssembledElement);
        ImplGeoPath :                   ifc_bounded_path ;
        ImplGeoPoints :                 LIST [0:?] OF ifc_cartesian_point;
        ImplGeoFrameProfile :           ifc_bounded_curve;
        ImplGeoPanelProfiles :          SET [0:?] OF  ifc_bounded_curve;
        ImplGeoTrimAProfiles :          SET [0:?] OF  ifc_bounded_curve;
        ImplGeoTrimBProfiles :          SET [0:?] OF  ifc_bounded_curve;
        ImplGeoPanelFrameProfiles :     SET [0:?] OF  ifc_bounded_curve;
        Height :                        OPTIONAL ifc_length_measure;
        Width :                         OPTIONAL ifc_length_measure;
        PanelThickness :                OPTIONAL ifc_length_measure;
    INVERSE
        FillsVoids :                    SET [0:1] OF IfcRelFills FOR
            FilledByFillingElement;
END_ENTITY;
```

Analogous Java classes for IfcFillingElement:

```
package IfcClasses;
import java.util.*;

public class IfcFillingElement extends IfcAssembledElement
{
    public IfcFillingElement() {

    // constructor internals here…

    }

    public IfcFillingElement(Vector vector)
    {

    // constructor internals here…

    }

    // accessors and modifiers here…

    protected ifc_bounded_curve     ImplGeoPath;                // 19
    protected Vector                ImplGeoPoints;              // 20 ifc_cartesian_point;
    protected ifc_bounded_curve     ImplGeoFrameProfile;        // 21
    protected Vector                ImplGeoPanelProfiles;       // 22 ifc_bounded_curve;
    protected Vector                ImplGeoTrimAProfiles;       // 23 ifc_bounded_curve;
    protected Vector                ImplGeoTrimBProfiles;       // 24 ifc_bounded_curve;
    protected Vector                ImplGeoPanelFrameProfiles;  // 25 ifc_bounded_curve;
    protected ifc_length_measure    Height;                     // 26
    protected ifc_length_measure    Width;                      // 27
    protected ifc_length_measure    PanelThickness;             // 28
    protected Vector                FillsVoids;                 // IfcRelFills;
}
```

Figure 5: IFC EXPRESS schema for `IfcFillingElement` and the analogous Java classes.

IFC EXPRESS schema for IfcRelVoids:

```
ENTITY IfcRelVoids;
    VoidsBuildingElement :    IfcBuildingElement;
    VoidedByOpeningElement :  SET [1:?] OF IfcOpeningElement;
    Interpenetration :        IfcAttLogical;
END_ENTITY;
```

Analogous Java classes for IfcRelVoids:

```
package IfcClasses;
import java.util.*;
import Utilities.*;

public class IfcRelVoids
{
    public IfcRelVoids() {
    }

    public IfcRelVoids(Vector vector) {

    // constructor internals here…

    //  set up the inverse relationship here...
        Vector  HasOpenings = this.VoidsBuildingElement.getHasOpenings();




    //  accessors, modifiers, and other methods here…

    private IfcBuildingElement  VoidsBuildingElement;
    private Vector              VoidedByOpeningElement; // IfcOpeningElement
    private IfcAttLogical       Interpenetration;
}
```

Figure 6: IFC EXPRESS schema for `IfcRelVoids` and the analogous Java classes.

EXPRESS schema for Accessibility Code component AccessibleElement:

```
ENTITY AccessibilityElement
    method :        IfcString;
    dimensions :    LIST [0:?] OF ifc_length_measure;
    comments :      LIST [0:?] OF IfcString;
    hyperlink :     IfcString;
    hypertext :     IfcString;
    ifcClass :      Class;
    level :         IfcString;
}
```

Analogous Java classes for AccessibleElement:

```
package AccessibilityClasses;
import java.util.*;
import Utilities.*;
import IfcClasses.*;
import CommentClasses.*;

public abstract class AccessibilityElement
{
    public AccessibilityElement() {
    }

    public AccessibilityElement(Vector vector) {

    // constructor internals here…

    }

    // accessors, mutators, and supplementary methods here…

    protected String getReferenceInformation(IfcElement element, int commentIndex) {
        return "<a name = \"" + element.getOwnerID().getIdentifier() + "." + commentIndex + "\">";
    }

    protected String getAnchorInformation() {
        return "See <a href=" + this.hyperlink + " target=\"code\">" + this.hypertext + "</a>\n";
    }


    public abstract void runRule(Vector elementSet, IfcElement element);

    protected String                method;
    protected ifc_length_measure    dimensions[];
    protected String                comments[];
    protected String                hyperlink;
    protected String                hypertext;
    protected Class                 ifcClass;
    protected String                level;
}
```

Sample instances of AccessibleElement in EXPRESS format:

```
#1 = ACCESSIBILITYDOOR          (
                                    DoorWidth,
                                    ((32., .IN.)),
                                    (),
                                    ../adaag/adaag.htm#4.13.5,
                                    Section 4.13.5,
                                    ELEMENT
                                );
#2 = ACCESSIBILITYDOOR          (
                                    DoorClearances,
                                    ((0., .IN.), (0., .IN.), (0., .IN.), (18., .IN.), (-60., .IN.), (80., .IN.)),
                                    (none, none, none, length, none, none),
                                    ../adaag/adaag.htm#4.13.6,
                                    Section 4.13.6,
                                    ELEMENT
                                );
```

Figure 7: Schema for the Accessibility Code component, the corresponding Java structure, and sample instances.

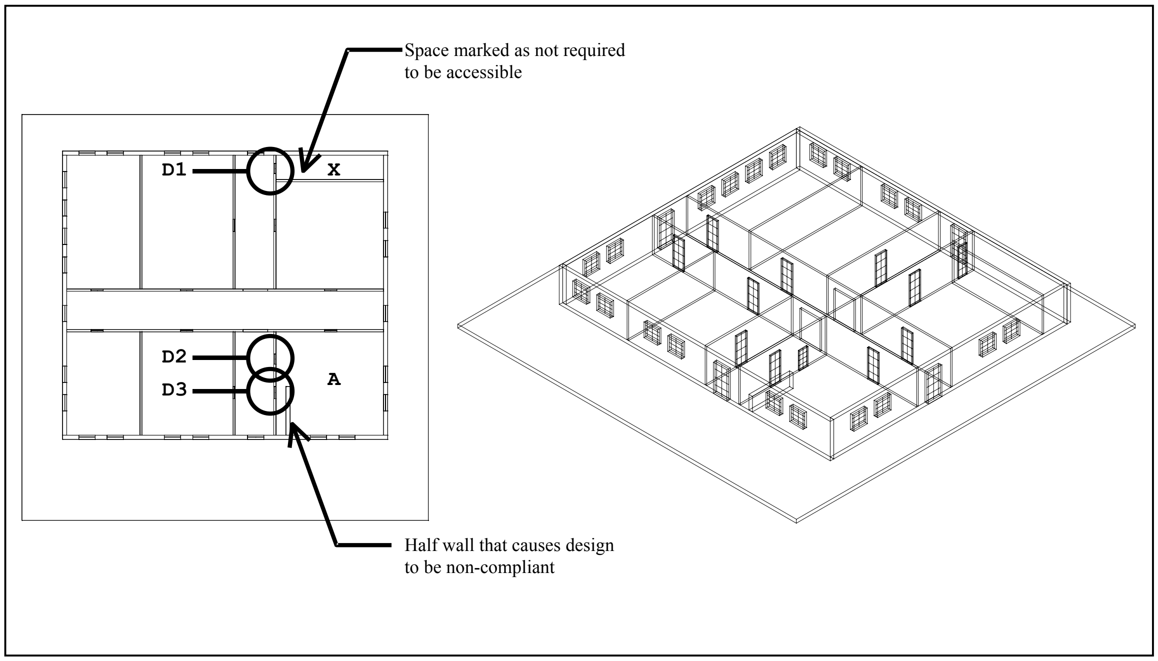
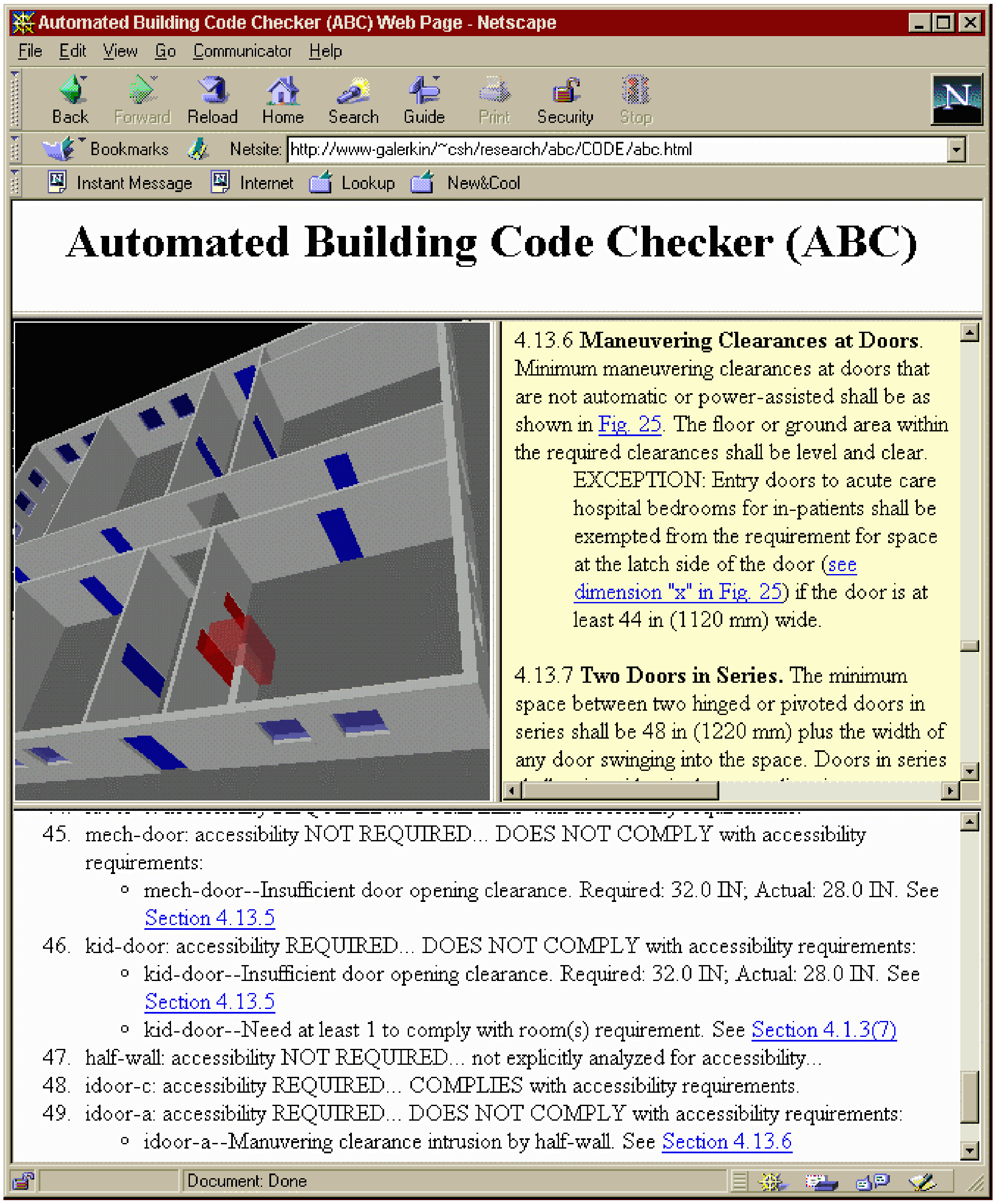

Space marked as not required to be accessible
D1
X
D2
D3
A
Half wall that causes design to be non-compliant

Figure 8: Plan and isometric views of the example design.


Automated Building Code Checker (ABC) Web Page - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Netsite: http://www-galerkin/~csh/research/abc/CODE/abc.html
Instant Message Internet Lookup New&Cool
Document: Done

Automated Building Code Checker (ABC)

4.13.6 **Maneuvering Clearances at Doors**. Minimum maneuvering clearances at doors that are not automatic or power-assisted shall be as shown in Fig. 25. The floor or ground area within the required clearances shall be level and clear.

EXCEPTION: Entry doors to acute care hospital bedrooms for in-patients shall be exempted from the requirement for space at the latch side of the door (see dimension "x" in Fig. 25) if the door is at least 44 in (1120 mm) wide.

4.13.7 **Two Doors in Series.** The minimum space between two hinged or pivoted doors in series shall be 48 in (1220 mm) plus the width of any door swinging into the space. Doors in series

45. mech-door: accessibility NOT REQUIRED... DOES NOT COMPLY with accessibility requirements:
    - mech-door--Insufficient door opening clearance. Required: 32.0 IN; Actual: 28.0 IN. See Section 4.13.5
46. kid-door: accessibility REQUIRED... DOES NOT COMPLY with accessibility requirements:
    - kid-door--Insufficient door opening clearance. Required: 32.0 IN; Actual: 28.0 IN. See Section 4.13.5
    - kid-door--Need at least 1 to comply with room(s) requirement. See Section 4.1.3(7)
47. half-wall: accessibility NOT REQUIRED... not explicitly analyzed for accessibility...
48. idoor-c: accessibility REQUIRED... COMPLIES with accessibility requirements.
49. idoor-a: accessibility REQUIRED... DOES NOT COMPLY with accessibility requirements:
    - idoor-a--Manuvering clearance intrusion by half-wall. See Section 4.13.6

Figure 9: Web page generated for the non-compliant design example.

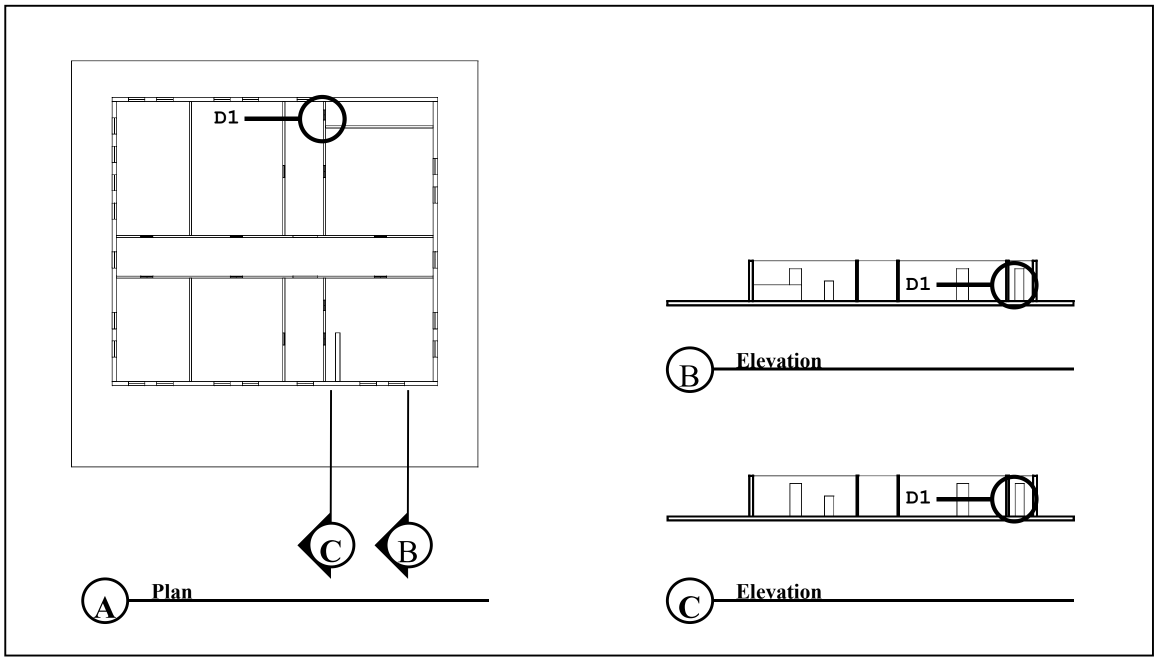
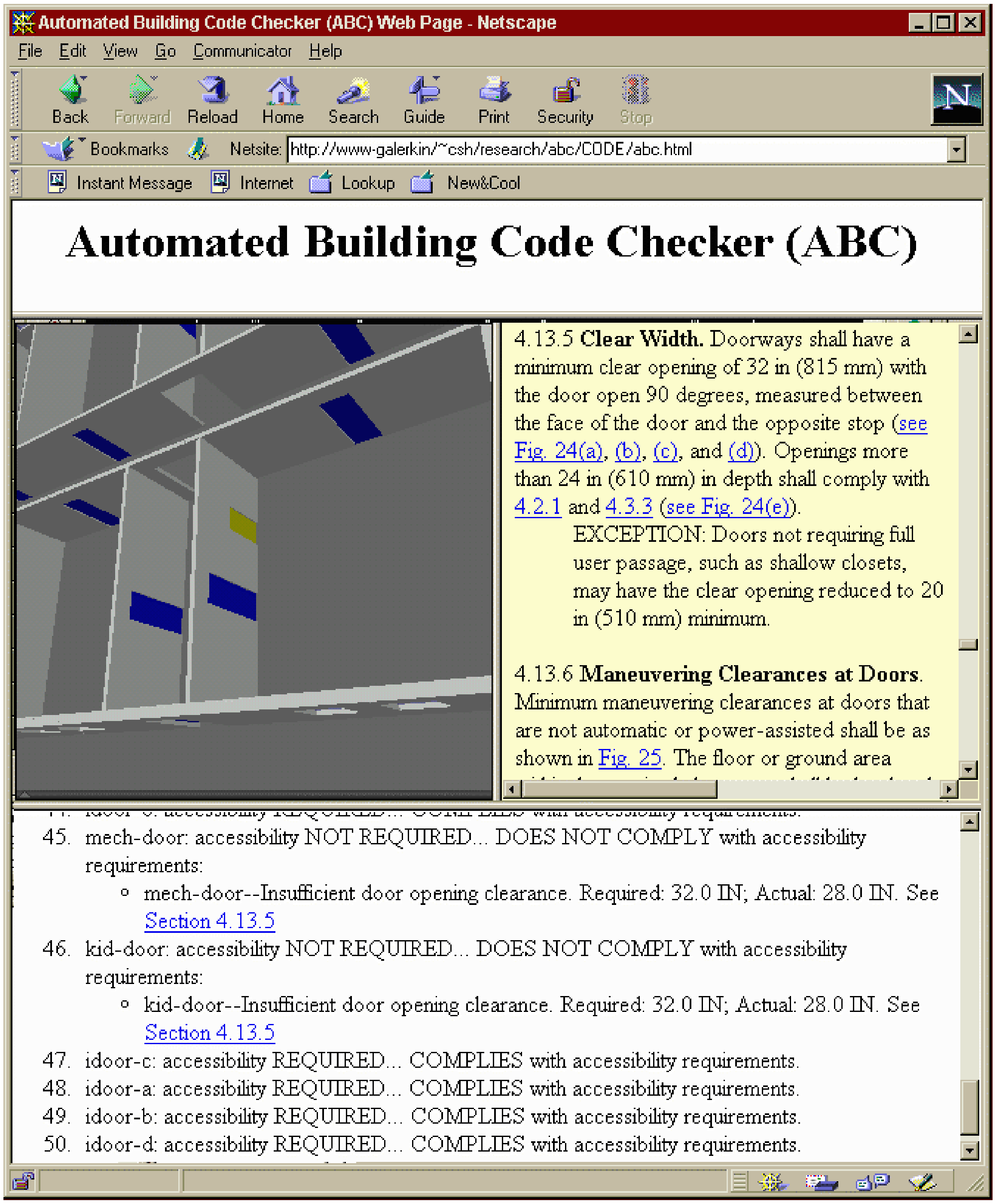


Figure 10: Two-dimensional representation: plan and elevation views.


Automated Building Code Checker (ABC) Web Page - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Netsite: http://www-galerkin/~csh/research/abc/CODE/abc.html
Instant Message Internet Lookup New&Cool

Automated Building Code Checker (ABC)

4.13.5 **Clear Width.** Doorways shall have a minimum clear opening of 32 in (815 mm) with the door open 90 degrees, measured between the face of the door and the opposite stop (see Fig. 24(a), (b), (c), and (d)). Openings more than 24 in (610 mm) in depth shall comply with 4.2.1 and 4.3.3 (see Fig. 24(e)).

EXCEPTION: Doors not requiring full user passage, such as shallow closets, may have the clear opening reduced to 20 in (510 mm) minimum.

4.13.6 **Maneuvering Clearances at Doors.** Minimum maneuvering clearances at doors that are not automatic or power-assisted shall be as shown in Fig. 25. The floor or ground area

45. mech-door: accessibility NOT REQUIRED... DOES NOT COMPLY with accessibility requirements:
    - mech-door--Insufficient door opening clearance. Required: 32.0 IN; Actual: 28.0 IN. See Section 4.13.5
46. kid-door: accessibility NOT REQUIRED... DOES NOT COMPLY with accessibility requirements:
    - kid-door--Insufficient door opening clearance. Required: 32.0 IN; Actual: 28.0 IN. See Section 4.13.5
47. idoor-c: accessibility REQUIRED... COMPLIES with accessibility requirements.
48. idoor-a: accessibility REQUIRED... COMPLIES with accessibility requirements.
49. idoor-b: accessibility REQUIRED... COMPLIES with accessibility requirements.
50. idoor-d: accessibility REQUIRED... COMPLIES with accessibility requirements.

Figure 11: Web page generated for the revised and compliant design.

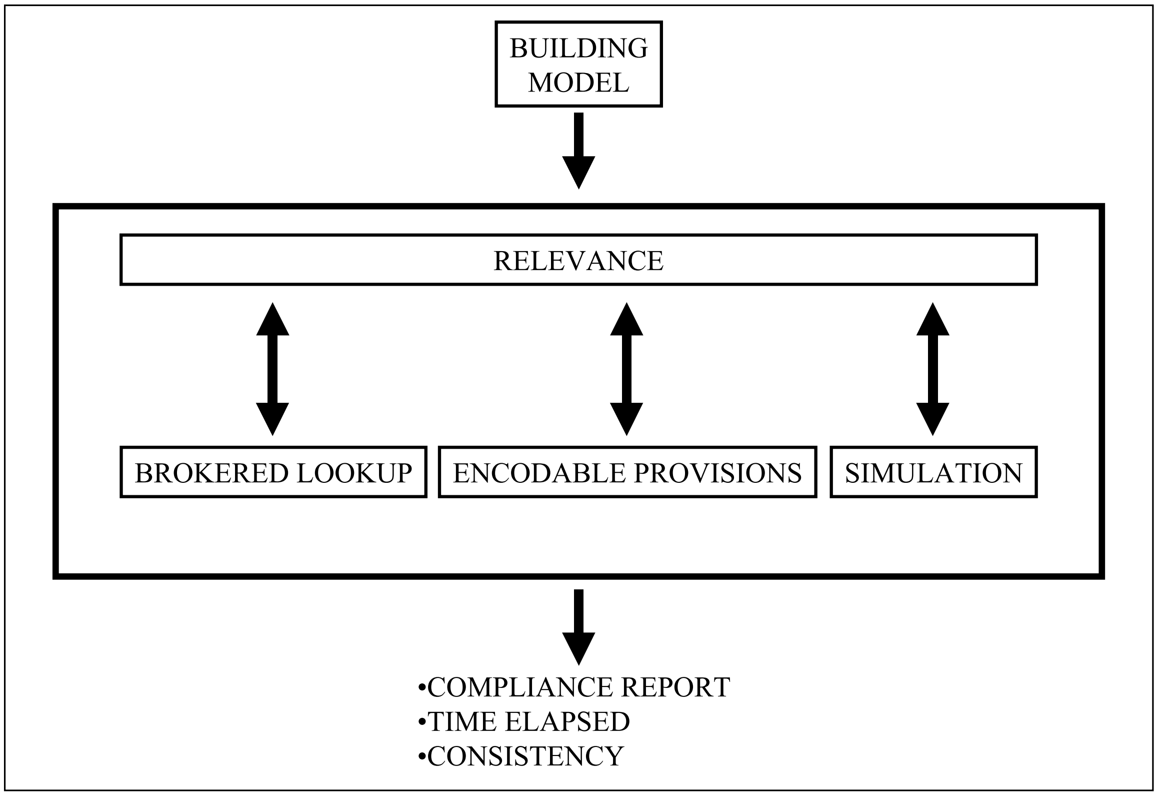


Figure 12: A proposed and enhanced code-checking framework.

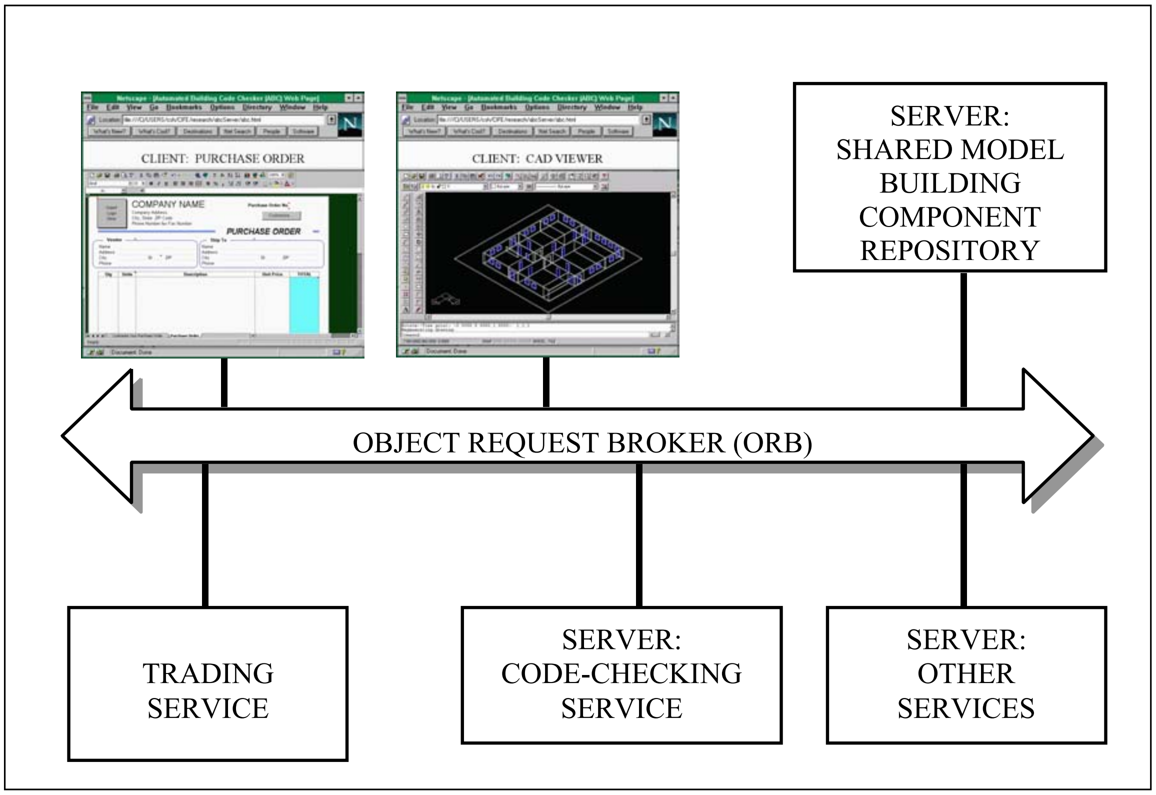

CLIENT: PURCHASE ORDER
CLIENT: CAD VIEWER
SERVER: SHARED MODEL BUILDING COMPONENT REPOSITORY
OBJECT REQUEST BROKER (ORB)
TRADING SERVICE
SERVER: CODE-CHECKING SERVICE
SERVER: OTHER SERVICES

Figure 13: A proposed shared model environment.

Accessibility Required?	Compliance?	Color
UNKNOWN	*	NOT red or yellow
REQUIRED	PASSING	NOT red or yellow
REQUIRED	FAILED	red
NOT_REQUIRED	PASSING	NOT red or yellow
NOT_REQUIRED	FAILED	yellow

Table 1: Building component color-coding rules.

Door	Accessibility Required?	Compliance?
D1	NOT_REQUIRED	FAILED
D2	UNKNOWN	FAILED
D3	UNKNOWN	FAILED
all others	UNKNOWN	PASSING

Table 2: Intermediate status information for the non-compliant design example.

Door	Accessibility Required?	Compliance?
D1	NOT_REQUIRED	FAILED
D2	REQUIRED	FAILED
D3	REQUIRED	FAILED
all others	depends	PASSING

Table 3: Final status information for the non-compliant design example.

Door	Accessibility Required?	Compliance?
D1	NOT_REQUIRED	FAILED
D2	NOT_REQUIRED	FAILED
D3	REQUIRED	PASSING
all others	depends	PASSING

Table 4: Final status information for the compliant design example.

List of Figures

List of Tables