An operating system includes means for interacting with a user, means for storing information, the information being organized using nuclear trees, means for interacting with underlying hardware, and means for initiating an application to run on the underlying hardware.
Figure 4

A

B - A

C

A - C

B - A
"The sky is blue" =

Figure 6

Figure 7

down the nuclear branch

IS

Applied Knowledge

blue

The

sky

The

sky

IS

down the nuclear branch
Average 5, 6, and 7

Figure 8A

A1

ADD

COUNT

DIVIDE

SUBSTITUTE

RESULT BRANCH #1

RESULT BRANCH #2

A nuclear command which results in replacing of question with result

Ref to children of "AVG" (NOW (H1))

This is a OR known to nucleus

A nuclear command reference to "ADD" branch result
Figure 9

Many had a little lamb
OPERATING SYSTEM USING DYNAMIC MULTI-DIMENSIONAL TREES

[0001] This application claims the priority to U.S. patent application Ser. No. 09/853,821, filed May 14, 2001, which is a continuation of U.S. patent application Ser. No. 09/064,824, filed Apr. 23, 1998, now U.S. Pat. No. 6,256,618, and which are hereby incorporated by reference.

BACKGROUND OF THE INVENTION

[0002] 1. Field of the Invention

[0003] This invention relates to operating systems, and, more particularly, to operating systems using trees as the basic data structures.

[0004] 2. Discussion of the Related Art

[0005] Conventional operating systems provide application interface software that enables applications to share the same basic 2-dimensional “look-and-feel”. But 3-dimensional modeling is left to the individual applications, and the operating system provides no cohesive structure into which applications can fit their 3-D modeling.

[0006] The result is that conventional operating systems provide no means for providing cohesion between applications. Two independent applications may have their own modeling structures, but, because they are not defined by the operating system, the models cannot merge.

[0007] For example, two independent software designers create a flight simulator and a virtual exploratory atlas, respectively. Conventionally, these would be separate applications. Conventionally, a user can merge the two only if the two software designers worked together.

[0008] Another issue is that of navigation. People have an innate ability to navigate, find things, recognize things, and operate within a three-dimensional environment. For example, people don’t memorize street names or maps in order to familiarize themselves with a new city; instead, as they explore, they gradually put together a mental image of the city. They don’t memorize lengths of streets; rather, they have a more primitive understanding of relative positions.

[0009] Navigating the conventional operating systems is like the memorization approach. There are hierarchies of folders and file names, coupled with a relatively tiny set of icons that do little to aid the memory and, sometimes, a difficult-to-use command language.

[0010] For example, a user is looking for a particular file. They have to ask themselves questions such as “what was its name?”, or “what folder did I leave it in?”.

[0011] As conventional systems grow, people lose their familiarity with the contents. Entire folders, and folders of folders, become alien, as people forget what they were, and what they contain. People do not naturally think in terms of hierarchies of folders, since it would be like trying to familiarize oneself with a town strictly by memorizing names.

SUMMARY OF THE INVENTION

[0012] Accordingly, the present invention is directed to an operating system using dynamic multi-dimensional trees that substantially obviates one or more of the problems due to limitations and disadvantages of the related art.

[0013] An object of the present invention is to provide a novel operating system paradigm and a universal infinitely extensible translation mechanism.

[0014] Additional features and advantages of the invention will be set forth in the description which follows, and in part will be apparent from the description, or may be learned by practice of the invention. The objectives and other advantages of the invention will be realized and attained by the structure particularly pointed out in the written description and claims hereof as well as the appended drawings.

[0015] To achieve these and other advantages and in accordance with the purpose of the present invention, as embodied and broadly described, in one aspect of the present invention there is provided an operating system including means for interacting with a user, means for storing information, the information being organized using nuclear trees, means for interacting with underlying hardware, and means for initiating an application to run on the underlying hardware.

[0016] In another aspect of the present invention there is provided a operating system including means for interacting with a user using nuclear trees, means for storing information, means for interacting with underlying hardware, and means for initiating an application to run on the underlying hardware.

[0017] In another aspect of the present invention there is provided an operating system including means for interacting with a user, means for storing information, means for interacting with underlying hardware, and means for initiating an application to run on the underlying hardware, wherein nuclear trees are used to control internal behavior of the application.

[0018] In another aspect of the present invention there is provided an operating system including means for interacting with a user using a Graphical User Interface, means for storing information, means for interacting with underlying hardware, and means for initiating applications to run on the underlying hardware, wherein the Graphical User Interface uses nuclear trees to integrate outputs of the applications.

[0019] In another aspect of the present invention there is provided an operating system including means for interacting with a user, means for storing information as applied knowledges, means for interacting with underlying hardware, and means for initiating an application to run on the underlying hardware.

[0020] In another aspect of the present invention there is provided an operating system including means for interacting with a user, means for storing information as applied knowledges, means for interacting with underlying hardware, and means for initiating an application to run on the underlying hardware.

[0021] In another aspect of the present invention there is provided an operating system including means for interacting with a user, means for storing information, means for interacting with underlying hardware, and means for initiating an application to run on the underlying hardware, wherein nuclear trees are used to provide an infinitely expressive language.
In another aspect of the present invention there is provided an operating system including means for interacting with a user using a Graphical User Interface and a general purpose translator, means for storing information, means for interacting with underlying hardware, and means for initiating applications to run on the underlying hardware, wherein the operating system permits different applications to communicate to each other using source trees.

In another aspect of the present invention there is provided a hierarchical structure for storing information embodied in a computer readable medium comprising a source for expressing a task, and applied knowledge to refer to at least one of an executable component and an interpretive component.

In another aspect of the present invention there is provided a method of interpreting input from a user comprising the steps of identifying a list of potential candidates to understand the input, recursively launching each candidate with the input, and evaluating results returned by each candidate.

In another aspect of the present invention there is provided a computer program product for operating a computer including a computer usable medium having computer readable program code means embodied in the computer usable medium for causing an application program to execute on a computer system, the computer readable program code means including computer readable program code means for interacting with a user, computer readable program code means for storing information, the information being organized using nuclear trees, computer readable program code means for interacting with underlying hardware, and computer readable program code means for initiating an application to run on the underlying hardware.

In another aspect of the present invention there is provided a computer program product for operating a computer including a computer usable medium having computer readable program code means embodied in the computer usable medium for causing an application program to execute on a computer system, the computer readable program code means including computer readable program code means for interacting with a user using nuclear trees, computer readable program code means for storing information, computer readable program code means for interacting with underlying hardware, and computer readable program code means for initiating an application to run on the underlying hardware.

In another aspect of the present invention there is provided a computer program product for operating a computer including a computer usable medium having computer readable program code means embodied in the computer usable medium for causing an application program to execute on a computer system, the computer readable program code means including computer readable program code means for interacting with a user using a Graphical User Interface, computer readable program code means for storing information, computer readable program code means for interacting with underlying hardware, and computer readable program code means for initiating applications to run on the underlying hardware, wherein the Graphical User Interface uses nuclear trees to integrate outputs of the applications.

In another aspect of the present invention there is provided a computer program product for operating a computer including a computer usable medium having computer readable program code means embodied in the computer usable medium for causing an application program to execute on a computer system, the computer readable program code means comprising computer readable program code means for interacting with a user, computer readable program code means for storing information as applied knowledges, computer readable program code means for interacting with underlying hardware, and computer readable program code means for initiating an application to run on the underlying hardware.

It is to be understood that both the foregoing general description and the following detailed description are exemplary and explanatory and are intended to provide further explanation of the invention as claimed.

BRIEF DESCRIPTION OF THE ATTACHED DRAWINGS

The accompanying drawings, which are included to provide a further understanding of the invention and are incorporated in and constitute a part of this specification, illustrate embodiments of the invention and together with the description serve to explain the principles of the invention.

In the drawings:

FIG. 1 illustrates the role of a kernel in the nGen system;

FIG. 2 illustrates the use of nGen as a translation paradigm;

FIG. 3 illustrates the 3-dimensional nature of nGen models;

FIG. 4 illustrates a hierarchy of the nGen model of FIG. 3;

FIG. 5 shows an example of how nGen model hierarchy changes as time evolves;

FIG. 6 is an illustration of a source tree;

FIG. 7 illustrates a 3-dimensional nuclear tree;

FIGS. 8A-8D illustrate an activation of a nuclear tree; and

FIG. 9 illustrates the use of nuclear trees in document creation.
DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0042] Reference will now be made in detail to the preferred embodiments of the present invention, examples of which are illustrated in the accompanying drawings.

Non-architectural Aspects of nGen

[0043] nGen is an operating system. It can run as an application under a host operating system, in which case it uses the host operating system’s application interface as its interface to the hardware. It can also run as its own operating system, in which case it interfaces directly with the hardware. nGen is also designed such that the base of the system could serve as the logical model for hardware design.

[0044] More generally, nGen is a machine independent operating system. It assumes that some kind of executable machine lies underneath itself, but it makes no assumptions about the nature of that machine. Therefore, it could be conventional software, actual hardware, non-binary based hardware, software machines such as neural networks, or even command-based machines (such as robotics).

[0045] In addition to organizing the running of software in the environment, nGen also provides a comprehensive language translation mechanism (redefinition) for any input or output language desired by the user. In particular, it allows for any subset of any natural language to be used as the input language for all purposes. Such a purpose is to translate a subset of natural language text input into executable code, where the meaning of “executable code” is dictated by the nature of the underlying machine. As such, nGen’s capabilities include serving as a system and application level programming environment.

[0046] Each nGen system is created by a parent nGen system. Thus, apart possibly from a small interface to a host operating system, the entire nGen system can be created using a subset of any natural language.

[0047] As illustrated in FIG. 1, a kernel is a small, fixed machine-dependent part of the system. It interfaces to the underlying platform (using a conventionally written interface, if required). The kernel implements the nGen architecture. The rest of the system, which grows dynamically, is composed of this architecture.

[0048] In addition to supporting the environment, and interfacing with the hardware, kernels are also responsible for interfacing with each other (even though the nature of their underlying machine may differ). The result is that two nGen systems, connected at the kernel level, would behave as a single, larger nGen system. It is intended that, in the preferred embodiment, all nGen systems would be part of a single nGen network. The nGen network would have the same characteristics as a single nGen system, varying only in the range of capabilities, and the amount of knowledge.

[0049] The manner of nGen distribution is as follows:

[0050] A parent nGen system is used as a development platform, thus increasing the capabilities and knowledge contained in the environment. The added capabilities and knowledge presumably reflect the “application” the developer wishes to create.

[0051] (b) The developer can then choose which parts of the parent to include in the new child nGen system.

[0052] (c) The parent then creates the child nGen system (which may run on a different underlying machine or operating system). If the user already has an nGen system, the new child and the existing system merge.

[0053] (d) As a result, a user who downloads separate nGen applications would end up with a single, seamless nGen system reflecting the combined capabilities of all the downloaded nGen applications.

[0054] The same basic arrangement can apply when a user visits an nGen-powered Internet site. If the user already has an nGen system, the user’s and the Internet site’s nGen systems would cooperate at the kernel level. If the user did not have an nGen system, the Internet site would create one for them.

[0055] The child nGen system may have programming capabilities (and, because the system provides a software development environment suitable for novice and expert alike, this is the general case). If so, the child nGen can create children of its own. The resulting distribution hierarchy provides the basic means for identifying particular nGen systems.

[0056] As noted earlier, a regular operating system cannot easily act as a virtual operating system, and attempting to create simulators (which are really limited virtual operating system) to run under rival operating systems is a difficult and error-filled process. With nGen virtual and regular are easily interchanged.

[0057] With nGen there is a relatively tiny hardware dependent component. Thus, an nGen system can be easily created for substantially different kinds of machines (like, for example, a neural network machine), and nGen running as a virtual operating system is designed to serve as a large operating system.

[0058] Conventional networks transfer files or similar units of information, and are seamless in the sense that a particular computer can use files from another computer.

[0059] With nGen, this seamlessness extends to the deepest level. For example, a thread of operation may go through several machines; this type of thread migration does not need to be explicitly specified, rather it happens spontaneously as the normal kind of operation within the system. This means that the level of cooperation is not dictated by decisions about units of information by software designers (i.e.—what is a file, and what is part of a file; or which logic constitutes a DLL, and which logic is merely part of a DLL, and therefore not shareable); the level of cooperation is dictated by the needs of the nGen systems. Furthermore, because the structure, utilization, and interpretation of content is consistent across all nGen systems, compatibility is ensured across systems. For example, a user in New York has asked a question involving atomic weights. His software “agent” (described below) has discovered that one way to answer the question is to first answer a series of subordinate questions, such as “what is the period of an indium electron at 150 degrees absolute?” In the course of experimentation (described below), other agents have been consulted, but none of them could answer this last question; however one of them suggested that particular humans, including profes-
sor Schmidt in Germany, might have relevant expertise. The user’s agent cooperates with the professor’s agent in conducting an on-line dialog with the professor (with the professor’s agent translating into German), in the hope of mutually deriving the answer. Above the kernel level, this has all transpired as if there had been no machine boundaries. More importantly, the two agents were designed independently, with no knowledge about the other; their on-the-fly cooperation occurred as a natural result of the infrastructure they were designed in.

Conventionally, software is created through one of the compilers for a particular programming language. There are several areas in which nGen improves upon this process:

With nGen, different languages are all processed via the same mechanism. For example, a software designer could use natural French and natural English as their language, and could use them interchangeably. Conventionally, if a designer wanted to use different languages, they would have to use separate compilers.

Ordinarily, different compilers or interpreters are needed for different purposes. For example, a person would use one language and compiler to program, a different piece of software with its own language for commanding, and yet another language and interpreter for querying. As illustrated in FIG. 2, with nGen, the language covers all purposes, so the user can stay with the same language, and never have to switch to another piece of software, no matter how varied the tasks.

Conventionally, if software is exported to a different kind of machine, it has to operate on a different platform. With nGen, the platform goes with the software. Even with JAVA applications (which theoretically are platform independent), variances in the recipient’s platform mean that the applications rarely work exactly as the designer intended.

Ordinarily, the operating system is distributed with the machine, and the user has to make a decision to switch operating systems (which is a risky operation, requiring a conscious decision). With nGen, users will generally see an nGen application first, and will not be aware of the base nGen system under the application. As they use more nGen applications or sites, their system’s nGen capabilities will gradually expand. If and when it becomes appropriate to consider making nGen the sole operating system, the user will already be familiar with nGen, and can make an informed decision.

Overview of nGen’s Architecture and Implementation

The highest level of the nGen system/network is nGenSpace. Within nGenSpace lie the nGen models. The nGen models are activated cyclically, to mimic a passing of virtual time. Some activity, such as movement, occurs over a span of time, while other activity, such as viewing, occurs at moments in time. The nGen models have nuclear trees, which provide the “brains” of the nGen models. The nuclear trees take care of translation and implementation, which together with basic actions such as movement and viewing, encompass all of a system’s actions. One particular nGen model of importance is the user’s agent, which plays a major role both in the user’s interface with nGen and in nGen’s internal organization. Structures of nGen models form the equivalent of both documents, and sites. The interpretation of these structures, usually by agents, takes on a large role within the system; this action is loosely labeled editing.

nGenSpace

As shown in FIG. 3, nGenSpace is a 4-dimensional (3 spatial, 1 time) virtual space, containing hierarchically organized nGen models. Usually, an nGen model occupies a contiguous and finite volume of virtual space. If it has (hierarchical) children, then the nGen models space encloses empty space, and the children are positioned within that space. If an nGen model has no children, then it is hierarchically a leaf. Otherwise, it is a non-leaf.

An nGen model can have its own 3-dimensional coordinate system, or, alternatively, may simply use the coordinate system used by its parents. If it has its own coordinate system, a set of simple and general conversion algorithms allow its position to be described to avoid collisions (a number of methods are known in the art to do this).

Similarly, an nGen model may define its own virtual time. Alternatively, it may use its parent’s virtual time. If it has its own, it must define its virtual time either as a function of its parents, or as a function of real-time. The outermost nGen model uses real time as its virtual time. Once again, simple conversion algorithms allow for rapid conversion.

As shown in FIG. 4, within the hierarchy of nGen models, any non-leaf nGen model implies a sub-hierarchy of nGen models. This sub-hierarchy is called an nGen model structure.

nGen models are activated on a periodic basis, with each activation being supplied an allocation of CPU usage. An nGen model’s parent may activate it for one of at least two reasons:

(1) to display itself, or to view its surroundings. These occur at moments in virtual time, to ensure that the snapshots capture a coordinated picture; and

(2) autonomous action. These occur over a range of time. For example, a virtual bird is activated at time t2, to simulate the action from the time t1 through t2. (Next, the nGen model will simulate the action from the time t2 through time t3, a subsequent time). Similarly, supposing that the virtual bird was in a virtual room (its parent nGen model); all peers of the bird (all other things in the room) would also be activated for the same range of time. The bird would do all movement appropriate for the passing of time; in addition, it may be doing background jobs, communicating with users or with other nGen models, or performing any other activity designed to be part of its continuous action.

During either of these two actions, an nGen model may discover messages from other nGen models in one of its mailboxes. If so, it deals with them accordingly.
An nGen model may also be activated on demand from another nGen model, for any number of reasons. In this case, the activation is immediate. The demanding nGen model’s action halts until the demanded action is complete. For example, a virtual bird may need to know exactly where the user’s agent is, in order to know where to move. Since it needs to know this immediately, it would activate the agent, asking it where it is. To discourage unnecessary use of this type of communication, the CPU time used is charged to the caller’s allocation.

As illustrated in FIG. 5, the hierarchy of nGen models is not static. It changes as models move around. There is no inherent limit to this movement. For example, a virtual bird may fly all over nGenSpace, passing through sites, documents, desktops, and public areas. In so doing, its position in the hierarchy changes. Similarly, a site, which forms a large nGen model structure, may float around nGenSpace. Not only does its position change, but because its (hierarchical) children go with it, so too do all of the nGen models contained in the site.

The Kernel in nGenSpace

nGenSpace is distributed across machines; but the architecture does not describe any such distribution. Rather, it describes how nGenSpace is distributed across the kernels that implement it. It is based on kernels, not machines. Where there is one kernel per machine (which may often be the case), this is either for convenience, or because the underlying hardware or operating system makes it necessary. It is not for architectural necessity.

Each kernel implements a portion of nGenSpace—its domain of nGenSpace (not to be confused with nGen model domain). In particular, each nGen model is implemented by one kernel (with one exception, about to be described). However, as nGen models move around, they may cross domain boundaries between kernels, and so transfer their allegiance. Kernel domains are dynamic. The exception just mentioned happens when an nGen model’s parent is under one kernel, and its child is under another. This nGen model has to appear simultaneously under both kernels: key information will be duplicated under the two kernels, but the majority of the contents of the nGen model will be under one kernel or the other. This may be illustrated with the following example:

A virtual city is located on a particular machine, under kernel “1.” Within that city lies a virtual store. The interior of the virtual store lies within kernel “2,” on another machine, so much of the store’s nGen model is appropriately held in kernel 2’s domain. The exterior of the store, however, is only of interest outside the store, on the first machine, not within the store on the second machine. To ask an nGen model under kernel 2 to display its exterior on the first machine would certainly be possible, but it would not be sensible. Therefore the store is represented under both kernels, 1 and 2, for displaying externally and internally respectively.

Responsibility for nGenSpace

Kernels implement key aspects of the architecture, either for security reasons, or to ensure consistency. With respect to nGenSpace, each kernel:

ensures that no child nGen model lies outside of its parents space;

takes care of the actual activation of nGen models, and ensures that no individual activation exceeds the allowance of CPU time specified by the parent of the activated nGen model;

implements the conveyance of messages between nGen models, and activates nGen models as necessary;

takes care of the actual movement of nGen models;

identifies and disallows overlap of nGen models;

Non-leaf nGen models also have certain roles, including:

(through the kernel) activating children nGen models, and allocating CPU time;

checking messages going both into and out of their area (for security reasons);

responding to attempted overlap within their space;

overseeing all viewing within their space;

In the example of two independent software designers creating a flight simulator and a virtual exploratory atlas, within nGen, these applications would be structurally mergeable. The result is that the user could be using his flight simulator to explore the virtual atlas. The two software designers, although working independently, are, by virtue of using nGen, effectively creating a single seamless product, since with nGen, all they need to do is use the nGen model framework.

Further, with nGen, navigation is accomplished via natural language (discussed below), and a consistent, point-of-view based 3-dimensional model. This approach parallels people’s innate ability to navigate.

For example, a user is looking for a particular file. Conventionally, they have to ask themselves questions such as “what was its name?”, or “what folder did I leave it in?” With nGen, they can ask themselves more intuitive questions: “where did I last see it?”, “what was it near?”, “where would I have put it?” In other words, they can approach the task in the same basic manner that they would approach real-life navigational problems.

In nGen, users build up a natural familiarity, a sense of where things are, relative positions, what areas look like, and, in short, a sense of feel.

Nuclear Trees

Nuclear trees are the brains of nGen models. They provide the raw, architectural framework upon which all real action occurs within the system.

As shown in FIG. 6, a source tree represents a thought of some kind (e.g., a request, a question, a statement, an integrated combination of thoughts). Note that a source tree corresponds to a self-modifying tree of the parent U.S. patent application Ser. No. 09/064,824, filed Apr. 23, 1998, now U.S. Pat. No. 6,256,618. Its nodes hold concept repre-
sentatives, which are references either to concepts in the real world (the concept of "dog", for example), or to things within nGen itself (the concept of a particular nGen model, or a particular user, or a particular kernel, for example). The arrangement of concept representatives within a source tree expresses some kind of thought (the concept of "the sky is blue" would consist of a tree containing the concept representatives of "sky", "blue", etc.).

**0098** An applied knowledge is a unit of knowledge that will be applied to a particular source tree (if the source tree represents a request, for example, the applied knowledge would contain the knowledge of how to implement that request).

**0099** When an nGen model is activated, it is the nGen model’s nuclear tree that gets activated. In the current implementation, every nGen model has a nuclear tree, and every nuclear tree is attached to an nGen model (although this arrangement could be deviated from, without upsetting the essence of the system—for example, a single nGen model can have multiple nuclear trees corresponding to it, or a nuclear tree can have no associated nGen model).

**0100** As shown in FIG. 7, a nuclear tree is conceptually 3-dimensional (logically, this has nothing to do with the 3 spatial dimensions of nGenSpace). It consists of source trees and references to applied knowledge. The source trees can be regarded as lying in planes parallel to one another. Connecting the planes of the source tree are nuclear branches.

**0101** Traversing a nuclear branch downward, one goes from an (upper) source tree root to a reference to an applied knowledge, then to a (lower) source tree root. The applied knowledge is that unit of knowledge that has been selected to process the reference in the upper source tree root. The lower source tree is where that processing takes place. The end result of that processing will be to alter the upper source tree; however, the process can continue recursively down the nuclear branches, and the upper source tree will remain in its original form until the recursion has unwound.

**0102** Activating a Nuclear Tree

**0103** There are at least two basic purposes to activation: translation, and implementation. Translation means taking a source tree and re-expressing it in some other form. Implementation means taking the source tree and implementing whatever it is expressing.

**0104** Both cases use the same underlying mechanisms, except that translation also uses a mechanism called refinement (which is addressed below).

**0105** There are roughly five types of implementation, depending upon the nature of the expressed source tree. The source tree may express a question ("what is 2+2?") a request ("move to the other corner of the room"), a statement (usually to be remembered or reacted to: "the sky is blue"), some kind of combination of these three ("the nearest store is 5 feet to your left; how long will it take you to get there?, and if it is less than an hour, go there"), or a structured expression of how to do something (usually to be remembered: "this is a way to average a bunch of numbers: take the numbers and add . . . "). The last case corresponds to conventional programming (at least in terms of functional-
Conventional parsing is usually done by an application (for programming languages, a compiler; for natural language, some natural-language “system”). Except to the extent that the application may contain references to other, specific applications or files, the rest of the knowledge on the machine is useless. For example, translation of material concerning landmarks may be occurring on a system in which there is a file containing the knowledge that the White House is a landmark. That knowledge may be important for disambiguating, but it will not be used by a conventional parser because there is no system-wide mechanism which will lead to the application of that knowledge. With nGen, the act of parsing draws off of the full knowledge of the nGen system, via the inherently recursive nature of the nuclear tree.

Conventional parsing is also generally rule-based. The parser implements grammatical rules. As a result, a conventional compiler implements a language which is a set of grammatical rules; these rules inevitably prevent extensibility, since natural language cannot be expressed as a set of rules. For example, C can be extended; but C could never be extended to express “what time is it?”, or “the sky is blue”, since these statements do not follow the grammatical rules of C. With nGen, there are no universal language rules; the rules are located within the individual applied knowledge, which may or may not be invented in any particular case.

In the case of natural language, conventional parsing is done via stand-alone applications which attempt to understand freely spoken speech. The problem generally is twofold. First, these parsers, being stand-alone applications, are not extensible. It is unfeasible to take a conventional parser and simply add entirely new ways to understand language; with nGen, this occurs as a natural result of the refinement framework. Second, conventional parsers provide no mechanism for certainty in the result. As a result, natural language parsing, as conventionally done, can never serve as, for example, a programming language. Programming with an unreliable tool could never result in reliable software. With nGen, the individual applied knowledge can insist upon 100% accuracy whenever that type of accuracy is needed.

In terms of code generation, nGen provides for any number of code generators to coexist. For example, the infrastructure can be extended to allow JAVA byte code generation, Pentium 2 executable code generation, C source language generation, SQL generation, or even generation of commands for robotics. In all cases, the same framework ensures that all available knowledge will be applied to the task of creating code, that entirely new approaches to code generation can be added, that all fit into the same framework.

As another example, the user asks a question “how many of my documents contain references to historical events?” This is translated into internal language. At this point, the system may have to translate different parts into different languages. Part of the question may end up in some proprietary language for text analysis, another part may get converted into SQL for database access, another part may be turned into JAVA byte code for analysis, another part into machine code for execution.

In other words, the refinement mechanism provides an infrastructure such that any number of languages (not limited to machine code) can be effectively utilized by the system, on behalf of the user. Without refinement, this process would become an organizational nightmare. Each of the different compilers would have to be called. All would have their own expectations about the input language. There could not really be any recursivity, because these compilers couldn’t work together. With nGen, this occurs as the natural result of the infrastructure.

Translation from human language to human language is another area of concern. Conventionally, stand-alone applications attempt to do the entire job. Refinement provides a mechanism whereby many different developers, each with their own area of expertise, can provide a piece to the puzzle.

For example, several designers add knowledge relevant to understanding French. These works come together in the refinement of French to internal language. Several other designers put together, independently, internal language to English generators. Some provide various general approaches (different top-level approaches), while other designers add more localized knowledge (such as when contextual information can be assumed, or which is the better way to frame a question, or which ambiguities are acceptable). Due to refinement, these diverse knowledges can co-exist whereas, conventionally, the software developers would have to coordinate their work in order to achieve the same effect. And even then, they would still not have a cohesive framework in which to organize their efforts.

The generating of natural language does not normally occur outside of stand-alone applications; certainly, it is not an operating system function. Within nGen, a general means for software to generate it exists.

For example, a particular application has to communicate with the user. Conventionally, the designer would prepare the questions in advance, and have them presented to the user as some sort of menu or dialogue box. If the nature of the question can only be determined at run-time, they have no effective means for creating the question. With nGen, the designer frames the question in nGen’s internal (base) language—or supplies the run-time logic for creating the same. Then, they simply arrange for the source tree to be translated into natural language. The refinement mechanisms ensure that if the knowledge to express the question exists somewhere on the system, that question will get expressed. The result is that developers can easily add run-time human-computer interaction, without specifying exactly how the computer generated language will be created—or even committing to any particular human language.

Implementation of Source Trees

As discussed above, another purpose of activating a source tree is for implementation. In these cases, the applied knowledge planted under the source tree will do what the source tree expressed. If the source tree expressed a question, then the applied knowledge will express a way to answer that question. For example, if the question was “what is 2x2?”, the applied knowledge may know of a way to add numbers together. If the source tree was a request such as “move 2 feet to the right”, the applied knowledge will know how to move nGen models. If the source tree was a statement such as “the sky is blue”, the applied knowledge
may know how to remember facts (generally, such applied knowledge will store those facts in the form of applied knowledge which knows how to answer the relevant question).

[0126] The infrastructure of refinement was used in translation because there are broad areas of similarity between the various refinement types. Implementation, however, does not show equivalent areas of similarity, so, in the preferred embodiment, it does not use refinement.

[0127] Often the applied knowledge will not know how to directly implement the source tree. Instead, it will rephrase the request, in the hope that applied knowledges exist that can then implement it. For example, "move 2 feet to the right" would first be rephrased as a sequence of questions and requests, such as "where exactly is this nGen model?", "which direction is to the right?" and so on. It is in this manner that complex requests are actually implemented by a relatively longer process of rephrasing, leading to a sequence of simpler source trees that can be directly implemented.

[0128] In practice, the use of implementation and the use of translation are seamlessly intermixed. For example, translation will result in many nuclear branches reflecting questions to be answered (implementations). Similarly, implementation is often accomplished through the use of translation; for example, one way to answer the question "what is the population of New Mexico?" is to translate the question into SQL (translation), and send the result of that translation to a database (implementation). Similarly, the principal means of remembering facts is to re-express them, and then translate them into applied knowledge (for example, "the sky is blue" will become applied knowledge which knows how to answer the question "what color is the sky?").

[0129] Implementation covers a wide range of activities. Command languages, data storage and search languages (discussed below) are some of the activities involved. More generally, the process of implementation covers the internal organization of software (akin to the handling of objects in an object oriented system).

[0130] Conventional command languages lead directly either to applications, or to interfaces to those applications that include a limited language associated with the operating system. These languages are usually parsed, after which actual implementation immediately follows. For example, in DOS, one finds the contents of a directory with a command like "dir c:", which, once parsed, leads to implementation by a standard utility. Such languages are inherently limited, because they can only express those things that the system directly knows how to do.

[0131] By contrast, with nGen, the language may express a request that is too complex to lead directly to the applications or utilities that can perform it. Instead, implementation provides a broad mechanism by which the request can be continually re-expressed, such that in one of those re-expressions the way to accomplish it naturally follows. More importantly, this language harnesses all of the knowledge within the system. Typically, a command language can only access a tiny fraction of the system's knowledge; with nGen, all knowledge is potentially involved in meeting the user's request.

[0132] For example, suppose the user asks the system to "find all documents which were created prior to the last board meeting, and which contain multiple references to the largest client". This "command" results in all knowledge within the system (and, through connection to the nGen network, all nGen systems) potentially being involved in meeting the request.

[0133] For data storage, nGen may utilize external databases (principally, but not exclusively, through formal language translation). In addition, nGen can be used to create in-house, conventionally structured databases.

[0134] Finally, and most importantly for this area of concern, nGen has its own approach to data usage and data entry. Traditionally, data is stored in records and fields (in a relational database). A record may contain information about a particular customer, for example, and the fields within that record may contain particular pieces of information, such as money owed and favorite product.

[0135] With nGen, the customer would be a concept representative. Information about that customer would not be stored as fields in that concept representative. Instead, information is stored as applied knowledge. For example, the knowledge that the customer owes $50 would be stored as knowledge that knew how to answer the question: "how much does this customer owe?"

[0136] It is important to note the following:

[0137] First, the answer does not have to be as direct as a number. The answer may be more subtle, as in "at least $20, but up until 2 days ago he owed nothing." The language (the concept representatives and the structure of source trees) allows such an expression to be made.

[0138] Second, in general, the attempt to answer may be one of many such attempts (as discussed with reference to experimentation). For example, several such applied knowledges may provide an answer, and the questioner may sort out which is most appropriate.

[0139] Third, the creation of the answer may be more subtle than just described. The customer representative may have stated that "this customer owes $50, unless they have an outstanding gift certificate over that amount" (which clearly needs to be stored as an applied knowledge, since it is primarily an expression of logic). What would ordinarily be unanswerable as a data entry, would in fact recurse the entire nuclear tree operation. This allows seamless mixing of raw information, and analytical software.

[0140] The rationale behind relational databases is that they more accurately model how people think about the "real world." nGen, however, allows to model more accurately how people really think about knowledge. Knowledge is not just the memorization of facts. Knowledge is an active part of a much bigger process. Certainly knowledge can sometimes be accurately modeled as a web of connected records; but this is rare: the vast majority of the time knowledge is an integral part of the thinking process, sometimes leading to new questions, and other times expressing a partial or subtle answer.

[0141] Object-oriented approaches to execution revolve around objects passing messages to each other. But the extensibility and expressiveness of the language in these messages is limited, in contrast to the source trees of an
nGen message, which represent an internal language with unlimited extensibility and expressiveness. Whereas specific objects are called to perform specific tasks, nGen has a general mechanisms for expressing requests (NOT who is to perform them), using this infinitely extensible and expressive language. “Language” refers to user→system, system→user, within systems, and across networks of systems.

[0142] The basic mechanisms provide for unfettered language growth, such that the language can be added to indefinitely, with no system-wide grammatical or syntactical limitations.

[0143] The resultant language can be capable of unlimited expressiveness (the best way to express this is that the language is capable of expressing any concept expressible in human language, with at least the same degree of accuracy and brevity as in human language).

[0144] The language can be extended to include any diverse way of expressing any particular concept. The best way to express this is that just as there can be many different and diverse human languages, all capable of expressing similar concepts, this language growth allows for similar diversity of appearance, structure, and content.

[0145] Thus, using nuclear trees and their associated uses (experimentation, refinement, etc., discussed below) can provide a language similar to human languages, which is unlimited in terms of diversity of appearance, and range of usability.

[0146] nGen adds at least two further features that have no counterpart in object-oriented programming (while an individual designer might simulate them, he would do this at great effort and in isolation, there is no general and coordinated mechanism for them):

[0147] (1) the general mechanism of experimentation (described below), and

[0148] (2) the fact that the expressiveness of the language allows a request to be rephrased, leading to exponential growth of the power of the system. For example, with object-oriented programming, the addition of a new object adds just that to the repertoire available to the designer (linear growth), whereas the addition of a new applied knowledge in nGen causes many previously insoluble requests to be rephrasable and hence implementable (exponential growth for both user and designer).

[0149] Experimentation

[0150] One of the ways in which an applied knowledge may operate (whether for translation or for implementation) is to experiment. The applied knowledge looks for other applied knowledges that may know how to meet the task at hand. Upon finding multiple candidate applied knowledges, these experimenters create a new nuclear branch for each of the candidates. The arrangement on the nuclear tree is as follows:

[0151] The experimenter has, above itself, the original source tree. If, for example, there are 10 candidates, then the experimenter will place 10 copies of that original source tree under itself, as 10 children of itself. Under each of these copies, the experimenter places a reference to one of the candidates. This way, each candidate operates in the usual way on a tree immediately above itself.

[0152] The experimenter will then activate each of these nuclear branches, either sequentially or in parallel, and use the results to solve the original task. “Using the results” can have a variety of meanings for example:

[0153] selecting one result
[0154] combining the results in some way
[0155] (in the case of implementing) deciding which experiment should be allowed to continue
[0156] replying “failed”

[0157] Any applied knowledge could be an experimenter, so the distinction between experimenter and non-experimenter is fairly arbitrary. However, it is assumed that most refinement types, and all implementations, will occur under the direction of an experimenter.

[0158] The net result of experimentation is an inherently non-deterministic system, with deterministic behavior being a subset. For example, in translation from text to internal language, candidate applied knowledges are invited to attempt to “see themselves” in the text. In general, either they don’t see themselves, or, if they do, they have to set up a recursion underneath themselves. Take “what is 4+4?” as an example. One (of many) candidates will recognize the “what is . . . ?” construct. But in order to say that it fully understands the text, it sets up a new experiment consisting of “4+4”, where it is looking for some kind of source tree that expresses a question. The virtue of experimentation is that it can allow many applied knowledges to be invited to understand any piece of text. The result is that the language, at any given point in time, is a function of the available knowledge, as opposed to a function of a pre-coded “parser.”

[0159] The principle of applied knowledge is that they operate by creating new source trees. No particular applied knowledge can know, in advance, if those source trees can be processed by the available knowledge. Experimentation provides the mechanism by which multiple approaches can be taken. Consider “the approximate population of New Mexico”. Many different applied knowledges will be invited. One of them might rephrase the question as a sequence including “what is the average population of counties in New Mexico?” and “how many counties are in New Mexico?” This particular approach might or might not work; but hopefully at least one approach will. Experimentation provides a generalized mechanism such that many independent nuclear branches can reflect the many ways in which a problem might be solved.

[0160] Applied Knowledge

[0161] All functionality within nGen is provided by applied knowledge.

[0162] Each new kernel starts with a set of previously constructed applied knowledges. These applied knowledges will have been created under the parent’s kernel, as will the bulk of the kernel itself (all but a conventionally written interface, if any, to a host platform), and then exported to the
new system. Included in the newly born system’s set of capabilities is the ability to create new applied knowledges. Either way, the process of creation is the same:

- **[0163]** (a) Each applied knowledge starts as a source tree containing the natural language text;
- **[0164]** (b) This source tree is then translated (with experimentation), into an internal form;
- **[0165]** (c) Optionally, further activation can be applied to produce executable code; and
- **[0166]** (d) The applied knowledge can be held in internal form, or as code, or both.

**[0167]** The internal form can be interpretively obeyed by the kernel. Internally, it contains concept representatives. Included in these concept representatives are nuclear commands, which are primitive actions that all kernels can perform directly.

**[0168]** The purpose of nuclear commands is two-fold: to provide security, and to allow kernels to directly interpret an applied knowledge.

**[0169]** To provide security, for example, the request to move an nGen model to a particular position in nGenSpace is a nuclear command. The reason is that all of the nGen models who own the space traversed have the right to block such movement. By ensuring that only the kernel can move nGen models, we ensure the integrity of any environmental security mechanisms.

**[0170]** To allow for direct interpretation, there are nuclear commands for cutting and pasting. A typical use of cutting and pasting occurs when an applied knowledge creates a new question: it will want to include in that question a portion of the source tree it is planted under.

**[0171]** The applied knowledge which is capable of multiplying two numbers together may be planted under “times 5?”. Its first operation is to ask “what is...?”, where “...” is equivalent to the first child of the source tree. In this case, the kernel sees the cross-reference, and plants below the applied knowledge the question “what is x?”. As another example, later in the same operation, the applied knowledge wants to reference the result of a prior operation. After activating the “what is x?” nuclear branch, through a nuclear command, the applied knowledge then wants the kernel to create a new question “what is... times 5?”, where “...” is the result of the “what is x?” nuclear branch (the “5” would have arrived in a similar manner).

**[0172]** At this point, the applied knowledge asks for that question, “what is 10 times 5?” to be activated. Presumably, although not necessarily, that would be handled by a different applied knowledge. As a final request, the applied knowledge asks for the result of that final nuclear branch to replace the source tree containing the original question with “the answer is...”, where “...” is the result of the “what is 10 times 5?” branch.

**[0173]** Through cut and paste operations, combined with primitive testing, blocks, and other similar constructs, the kernel “interprets” the applied knowledge, primarily by extending the nuclear tree downward.

**[0174]** The creation of executable code involves taking the interpretive applied knowledge, and creating an executable equivalent. This includes simulating the action of the kernel, deciding in advance on applied knowledge invitation, bypassing experimentation where possible, and optimizing as appropriate.

**[0175]** Multi-threading occurs at two levels within nGen. At a higher level, the nGenSpace infrastructure implements a “top-level” multi-threading among nGen models, through the cyclical action described earlier. At a lower level, individual applied knowledges may be multi-threaded internally. For the interpreted case, there is a nuclear command to take a block of requests and implement or translate them sequentially or in parallel as resources allow. For the executable case, such blocks are coded as multi-thread operations. In both cases, the kernel grants CPU allocations to each thread, and handles those situations where CPU-time has expired, re-entering the interpreted tree or the executable code upon the next activation.

**[0176]** An example of multi-threading within the applied knowledge involves translation of text. This is a heavily experimentative process, with many experiments being run in parallel, each with sub-experiments of their own. The kernel, in these cases, can divide the CPU time among the nuclear branches, thus multi-threading the process. As a side note, due to thread migration, these threads can be implemented on a different machine. In other words, one aspect of nGen is that parallel processing can be implemented across machine boundaries—or, on separate CPU’s in a multi-CPU chip.

**Organization**

**[0177]** Regions

**[0178]** Applied knowledge and concept representatives are grouped into regions. In the first place, they are grouped into a hierarchy of base regions, where no two base regions can overlap. Super regions can be formed indiscriminately as groups of base regions, primarily as a convenience for referring to them. During the activation of a nuclear tree, some regions are open, while others are closed. A closed region cannot be opened if there is no knowledge about it in a currently open region.

**[0179]** Regions have many different purposes. Regions allow multiple nGen models to share information in a practical manner. For example, there may be millions of nGen models in a particular environment, each having its own knowledge. Regions provides a mechanism to ensure that the total volume of knowledge that is shared among them remains scalable.

**[0180]** Regions also provide a convenient means of organizing information. For example, a particular agent may know of many agents which specialize in physics. The physics agents may have areas of specialty, referred to as regions. In addition, there can be knowledge about the various regions. The first agent may use this information to decide which other agents to bring into a particular task.

**[0181]** For example, the physics agent knows 1000 agents. It knows that agent 50 has a specialty in regions x and y. The physics agent knows that region x covers quantum mechanics, and region y covers relativity. The physics agent, on behalf of another agent, is attempting to answer a question that involved both subjects. Because it knows this information, the agent knows that agent 50 is a good agent to consult.
Regions provide a convenient means of distributing information. A newly created nGen model will inherit certain regions, giving it a solid base language and base set of capabilities. A "document", discussed below, will be sent with regions of relevant information.

Regions also provide a solid component of nGen's general security. All nuclear tree manipulations are handled by the kernel. For all intents and purposes, knowledge in closed regions is inaccessible. In addition, all inter-nGen model communication, all intra-nGen system and all inter-nGen system communication, is handled by the kernel, which refuses transfer of currently closed regions. Moreover, regions can have security levels. Regions that contain disk-access capabilities will be in a more secure region than regions which contain general knowledge.

For example, a user decides that certain tax information is private. By placing it in a region, and disallowing transfer of that information, the user can be protected against malicious applets accessing the information, as well as unintentionally divulging the information to other parties.

Communication

ID's for referring to nGen models, applied knowledges, concept representatives, and nuclear tree nodes are the responsibility of the local kernel, and are called local ID's.

To allow communication between kernels, each of these entities may have a universal ID: in which case every kernel that recognizes this entity has its own local table entry giving universal-to-local equivalence. Presence of this table entry implies that the corresponding local entities under each kernel are the same. For example, if two kernels each have a table entry for the internal equivalent of "it", this implies that "it" means exactly the same under both kernels, even though their local ID's are different. To avoid duplication of universal ID's, each kernel is given a unique kernel ID (which may be a function of its creation hierarchy), and each universal ID has, as one of its components, the kernel ID of its creator.

If two software developers decide that two independently defined entities are, in fact, the same, this is expressed by giving them both the same universal ID. For example, two developers on two different machines independently add the concept of "dog". These will be treated as two separate concepts, each with its own local ID. However, someone, having identified the equivalency, can declare them equivalent by giving them a shared universal ID, causing their respective kernels to treat them as the same concept.

All communication (user-to-system, user-to-user, system-to-user, inter-system, and intra-system) occurs via mailboxes of the nGen models (with the exception of the user's direct communication with their agent). For example, two people carrying on a conversation will be implemented by mailbox communication between their respective agents. In cases where the communication contains universally recognized entities, the respective kernels translate them to local ID's for internal use, and to universal ID's for transfer.

nGen's mailboxes, combined with the action of the kernels, provide a comprehensive mechanism for communication. Sometimes, the communication may be executable, in which case it parallels applets. At other times, the communication may be nGen models, in which case it parallels document formats, 3-D modeling formats, and hypertext. At still other times, it may provide user-to-user communication, paralleling e-mail, bulletin boards, and chat mechanisms.

Agents

One particular nGen model under each kernel is the agent (as used conventionally, a vague term here, used in a specific sense as discussed below). The agent plays a special role within the environment (with the domain of a single kernel being an environment). It is the user's principal interface to their system; and it controls the devices. (In the nGen operating system case, "controlling the devices" includes the drivers; in the virtual operating system case it means interfacing the host operating system's access to the devices.)

What users see on their screens is determined by their agents. Agents may look in a particular direction, displaying on the screen what they see; alternatively, they may ask other nGen models to look, taking the results and putting them on the screen. For example, the user's agent moves to the left; the user's screen would reflect that change in position. Alternatively, the agent has a virtual camera set up in another area of nGenSpace; the agent then places the pictures from that camera on the screen.

Similarly, the agent receives all user input. Keyboard input is received by the agent. The agent could process the input itself, presumably through translation and implementation; alternatively, it could pass the input to another nGen model for processing. Either way, the agent has the first opportunity to deal with input from the user.

Mouse input is likewise handled by the agent. The agent may use a separate nGen model to represent the mouse in nGenSpace, but all input goes directly to the agent who, in most cases, moves the mouse nGen model accordingly. For clicks, the agent will generally send a message to the clicked nGen model, or it may process the click itself.

All other devices, either input or output, are handled by the agent in a similar manner.

The agent may allow other nGen models to use devices; but the kernel only permits device driving by nGen models that have been so allowed.

The agent takes on much of the responsibility ordinarily associated with an operating system. For example, there are no such things as "files" within nGen. Instead, there are nGen models, and regions of knowledge about those nGen models. The agent has exclusive (and temporarily transferable) right to move nGen models around, according to either the user's direction, or to its own logic. Thus the equivalent of "file management" is taken care of by the agent.

Similarly, the agent dictates how CPU time is allocated for all nGen model structures contained in the environment. Thus the top-level multi-threading is effectively controlled by the agent.

Being in control of input and output, the agent is the eyes and ears for the user. It will usually have the richest language, and the largest reservoir of applied knowledge, of all nGen models in the environment. In addition, it will have
its own virtual rooms, for viewing documents and for screening content. In short, the agent takes care both of the user, and of the user's information, content, and desktop.

[0200] The Agent as Representative

[0201] nGen does not need to make a distinction between client and server: all nGen systems are capable of being both. When receiving visiting proxy agents, the user's agent has full security over the environment, and also interacts with the visiting agent.

[0202] For example, John Smith’s agent visits the environment of Jim Jones. In order to navigate, the agent of Jim Jones works with John Smith's agent, through communication (including thread migration and the lending of regions). In some cases, John Smith will be visually exploring, such as wanting to see a virtual art gallery in Jim Jones’ environment. At other times, the proxy agent of John Smith will simply be visiting in order to ask Jim Jones’ agent for assistance in solving some questions from John Smith. In all of these types of cases, when John Smith visits the Jim Jones’ environment, it is the agent of Jim Jones which represents Jim Jones himself.

[0203] The Agent as Environment Navigator

[0204] One of the principal functions of an agent is to act as its person’s navigator within their environment. In this function, the user expresses what they want to find, using natural language and the language; and the agent utilizes its own applied knowledges to understand the user, find the relevant material, and generally assist the user in navigation.

[0205] Conventionally, navigation at the operating system level is done through hierarchical arrangements of files. Optionally, small amounts of information can be held about files, such that the user can utilize a structured query language to find particular files. Navigation at the lower level is application specific. The user can open a file such as a database, and use the database’s search mechanisms for navigating within, for example.

[0206] With nGen, the navigation is all handled by the agent. The same language and mechanisms are used to find content of all types. For example, the user may ask “find a way to average two numbers together”. The agent proceeds to look for applied knowledge that meets those criteria. In this case, the agent would set up an experiment, and instead of activating the nuclear branches, examine the ones that responded. In examining them, it would determine which are re-expressing the problem, and which solve the problem directly; the latter can be handed back to the user (in natural language “take the numbers and add . . . “, or as a reference to the applied knowledge).

[0207] In another case, the user may ask for “all documents that contain references to historical figures from the 18th century”. Here, the agent is having to implement several lookups. It will be consulting local areas of information to find relevant lists; it will be looking through the environment for possible documents; it will be running analyses on which nGen models are really documents; it will be running analyses on whether the documents contain applicable references. But in all cases, the agent is acting as the user’s personal navigator, so the user does not have to go through all of these actions himself.

[0208] In all of these examples, the user’s agent could have been acting instead on behalf of another user visiting the environment. For example, it may have been another user’s proxy agent who was asking these question on behalf of its own user (subject, of course, to the local agent’s knowledge of which regions to keep closed, and which spatial areas are off-limits).

[0209] The Proxy Agents

[0210] By definition, agents stay in the user’s environment (they need to be there, since they are responsible for that area). In those cases where the user wants to visit other areas of nGenSpace (the equivalents of sites, and other desktops), the agent sends out a proxy agent. A proxy agent is an nGen model created by the agent, and, unlike the agent itself, is able to go into other areas of nGenSpace.

[0211] Proxy agents can serve many different functions, ranging from autonomous exploration, taking the user through nGenSpace, to finding content, viewing content, or gathering knowledge. In all cases, the proxy agent serves as the user’s representative.

[0212] For example, a user wishes to visit an nGen site. His agent creates a proxy for him. The proxy inherits all of the agent’s knowledge, except for knowledge specific to the user’s environment. This proxy is given whatever appearance the user wishes, including a human-like representation. The agent displays on the screen whatever the proxy sees. So if the user goes to an nGen “site”, they will see all of the other proxies visiting the site.

[0213] This site may be a baseball oriented site. The site’s agent may have a large reservoir of baseball knowledge, so the user can (through his agent, to the proxy, to the site’s agent) carry on conversations about baseball. Alternatively, there may be virtual games on this site. In these cases, the user’s proxy would serve as their point of view for the baseball game, relaying all input and output through the user’s agent.

[0214] Multiple users can, through these proxies, communicate. For example, two users may share baseball videos, through the respective agents, or through the proxys. If one user alters the site in any way, such as if there is a virtual doorway, and the user opens it, all other users will see the movement, since they are all sharing the same area of nGenSpace.

[0215] The site’s agent may be able to act as a portal. It may know of a number of other baseball sites, and be able to tell the user about them. The agent controlling a portal might have large amounts of information about the various destinations, allowing it to carry on natural language conversations about them, with the visiting users. In addition, the site’s agent can interact with other sites’ agents, in order to answer questions in a more dynamic manner.

[0216] In all cases, the spatial environment is providing a consistent framework for the content. It is the agents’ knowledge about that content that allows the natural language interaction. The applied knowledge is a natural outcome of designing the site with nGen. Information can be added dynamically as needed by the site owner, to constantly augment this capability; for example, the owner sees that a particular video of a game also has some rare footage of a former president. By telling his agent that information, this
information can then be dynamically applied when a user, perhaps on another site, asks if there is any known video of FDR attending a baseball game.

[0217] In addition to navigating sites, proxies can also act as personal search agents. The user asks their agent to send out a proxy to scour the Internet in search of baseball facts, or baseball videos. The proxy agent then proceeds to wander around nGenSpace collecting what the user wanted.

[0218] Alternatively, the agent may use a proxy to set up a virtual recording camera. The user may tell their agent that they wish to get a recording of a streaming broadcast of a particular band. Their agent would send out a proxy to watch and store the broadcast.

[0219] Agents as Networks

[0220] Agents can be designed to serve as localized networks. Similar to the region structures, groups of agents would be set up as base groups, and superset groups (in which case the groups are non-overlapping, and overlapping, respectively). The group's regions may be different from those of the individual agents within the group; and when an agent is acting as the representative of its group, it will find its group's regions open, not its own.

[0221] For example, five people create a group out of their agents, all centering around baseball. These agents, acting individually, will be representing their respective users. But when these agents act as a group, they will be representing the group itself. One of these agents may contain a large set of knowledge that is used when representing their user (it may contain a mixture of baseball, and other interests). But if a question is asked of the agent as a member of this baseball group, a different set of knowledge may be used (it may contain a larger set of baseball, and nothing else).

[0222] This process allows groups of agents to have different characteristics than their component agents. It is similar to a business. As employees, people have different behaviors, utilize different knowledge, and perhaps have different appearances than they do when they are not working.

[0223] Agents as Search Engines

[0224] Agents can have particular roles within nGenSpace. A particular agent may act as a public navigator. It might send large numbers of proxies out in order to collect as much information about nGenSpace as it can. It can then, in turn, sell these services to users.

[0225] Conventional search engines utilize a natural language interface, which generally is hit-and-miss, incapable of real understanding, and is useless for attempting to express complexity. nGen utilizes the language as described previously, a mixture of naturalness, expressiveness, and accuracy. Unlike nGen agents, conventional search engines do not display any real understanding of either the content they are representing, or the user's request. Improvements like XML allow for a slightly greater knowledge of content, but they fall well short of the kind of understanding provided by nGen's processes.

[0226] For example, the user asks to find all areas of nGenSpace that have baseball trivia, and also a number of trivia-based games. The language understanding ability of the system allows such a request to be fully understood by the "search engine" agent (which was consulted through the user's agent). With all content there is also applied knowledge about the content. For example, a particular site may have some baseball trivia content and some competitive games. The search engine agent would ask the site whether those games are primarily trivia-based. Because the site's agent would have a rich repertoire of applied knowledge about its own environment, and because it can interact with other agents in order, for example, to determine exactly what "trivia-based" meant, it could answer that question.

[0227] It is this mixture of factual knowledge, analytical tools, expressiveness of communication, and the recursive nature of the process, which allows the site to have a much larger effective knowledge about the site, than just the set of facts that they have stated.

Editing

[0228] "Editing", refers to a broad subject that covers both creation of content, and the interpretation of content, where content consists of nGen models and, at times, associated applied knowledges. A subset of this is conventional editing of documents, but more broadly, it covers all types of visual content. (Note that a document can be a text document, hypertext, can include images, multimedia, can be icon-like in nature, a web page, or can be a part of a larger structure of related documents.)

[0229] Any nGen model structure can model a physical entity, and have the necessary knowledge to display itself. In addition, it may also have additional knowledge, relevant to itself. Finally, any other nGen model may also have knowledge about any other.

[0230] When any content is transferred from nGen model to nGen model (from here on, it is assumed that agents are doing the bulk of such transfers), regions of knowledge may be added in with the transfer of nGen models (regions over and above the regions contained in the nGen models). These transferred nGen models are directly viewable, and the agent can, without any additional applied knowledge, manipulate them (move parts around, delete parts, merge them with other nGen model structures).

[0231] Anything can be covered in this basic mechanism. It may be a complex physical model of an exploding bomb, a virtual game, a document, or even just a single character of text. Conventional editing, the display and manipulation of text, is a subset of these mechanisms.

[0232] Conventionally, a short document consisting of "Mary had a little lamb" would consist of a linear sequence of text ("Mary had a little lamb") interspersed with codes (usually paired, as in "begin italic", and "end italic"). This approach has the advantages of size (using codes that have ranges lessens the total space for a given document), and simplicity (at least for simple cases).

[0233] In the simple case, under nGen, the document would appear the same (to the user), but the underlying "coding" would be substantially different, as illustrated in FIG. 9.

[0234] The "stored" form of the document would consist of 19 nGen models arranged in a simple hierarchy. The root node would consist of an nGen model representing the page
(or whatever was appropriate). Under this node would be 18 nGen models, one for each of the 17 characters, and one for the underline.

0235] These 19 nGen models, cumulatively, would be able to display themselves such that the result would be indistinguishable from the appearance within a conventional word processor. The difference lies in how the agent “reads” the document. There is no coding within nGen’s document, as such. Rather, the agent interacts with the nGen models, to ascertain the document’s structure.

0236] For example, one of the nGen models is modeling the “h” in “had.” There would be a small conversation by which the agent would find out about any formatting:

0237] Agent: “Are you text, or a picture?”
0238] nGen model: “Text”
0239] Agent: “How many letters?”
0240] nGen model: “One”
0241] Agent: “What letter?”
0242] nGen model: “h”

0243] Agent: “Are you italicized?”
0244] nGen model: “Yes”
0245] (and so on).

0246] Note that the process can be word-based, or paragraph based, or sentence based, although it is believed that the character-based structure, as described above, is optimal.

0247] As a result of this interaction, the agent can construct an understanding of the formatting within the document. There is no inherent limit on the scope of the conversation.

0248] Once the agent has an understanding of the document, it can then store the resultant information in many ways, including traditional linear sequences of code and text, or hierarchical models of formatting controls (of course, it does not need to do this, since it can always go back to the nGen models and interact with them again).

0249] The agent does not necessarily have to know all the details within a document. For example, it does not need to know that the “h” is italicized if the user is just viewing the document. In addition, the applied knowledge carried with a character-sized nGen model would be grouped into regions, which would be shared by many nGen models. Finally, the interaction can be made more efficient through list-functions (i.e., “do you have any non-normal characteristics?” → “Yes, I am italicized and bolded”).

0250] This nGen model structure would be completely self-sufficient, able to display itself without any assistance from the agent. But there would be no information about the logical structure of the document. For example, there would be no information about which characters form words, where the sentences are, which page breaks are user-defined, and so on.

0251] This leads to another main subject: deriving logical structure via the utilization of refinement and implementation.

0252] The agent receiving a document would attempt to find out the document’s logical structure. Taking a simple example, it may wish to know how many words are in the document. It would accomplish this by raising a question “How many words are in this document?” (utilizing the logic nodes and concept representatives to form the question). This question could go anywhere. In the simplest case, the sending agent (the agent which actually created the document) would supply the necessary applied knowledge to answer the question. In the more complex case, the editing agent would utilize all available applied knowledge (and other nGen systems) to attempt to come up with a way to answer the question. For example, another nGen system may have analytical tools which can accurately determine where words are, based on the position of the characters, while yet another may have tools which can determine whether unbroken strings of text are actually mathematical equations, and so on. Through the action of refinement, spanning across machines, if necessary these diverse tools can come together to assist the user’s agent in determining the logical structure of the document.

0253] The result of this is that the formatting language, the means of expressing the format, and the means for deducing format are all infinitely extensible, and draw off of the full knowledge of all nGen systems.

0254] Finally, any document structure modeled conventionally must utilize a fixed “language”. This may be a mark-up language, or a Word-type format. Either way, the language has, at best, very limited extensibility. Because of the seamlessness derived by nGen’ approach to documents, nGen documents are not similarly constrained. The following examples of what could be expressed show the range of the results of this kind of seamlessness:

0255] The document writer wishes to underline every word that is a reference to any event occurring after 1940.

0256] He wishes to have the “h” in “had” invert itself at 12:00 every day, so long as the word “nGen” appears in the document.

0257] He wishes the displaying editor substituted synonyms appropriate for the age of the viewer.

0258] He wishes paragraph breaks to be inserted at the best point, whenever there are more than six sentences in the paragraph.

0259] He wishes the best available portrait picture of the President to be placed to the right of a particular paragraph.

0260] In each of the above cases, the agent, or the individual characters, would utilize the refinement process, which could then lead to the application of knowledge swrawn throughout the system and the network of systems. From the designer’s vantage point, all they are doing is expressing what the receiving agent is to do (in the first case, the instructions would be “underline every word that is a reference to any event occurring after 1940”). If the designer felt it necessary, they could optionally attach logic (adding it to the pool of applied knowledge sent with the document), or they could rely on the receiving agent to come up with a way to accomplish the task.
The act of creating content within nGen is similar to the act of searching for content. The user states, in natural language, what they want to create; through implementation and translation, the agent proceeds to create it for them. As a very simple example, the user wishes to type an "e" after the "h" in "has". The user could state "place an 'e' after the 'h' in 'has'". The agent then uses a blank nGen model, and adds the necessary applied knowledge to it. It then figures out its exact dimensions, and adds it to the page nGen model. It then proceeds to move the other nGen models as necessary.

Alternatively, the user is in a mode whereby all keyboard input is treated not as text to be understood, but rather as text to be placed wherever the cursor nGen model is. In these cases, the agent may treat each keystroke as an abbreviation for a complete sentence (a sentence like "place . . . has" in the previous paragraph), and then process that sentence through recursive implementation. The reason for doing it this way is to make use of the full power of that process. It may be that new knowledge has arrived which provides a new way to add text, one which, for example, does a better job of rearranging the other nGen models. The implementation mechanism provides the basic means by which that additional applied knowledge gets applied to the task at hand.

Conventionally, content creation tools are point-and-click based. The user is given very little real power, since their means of expression is so limited. With nGen, on the other hand, the user could, for example, ask their nGen agent to turn the text into pig-Latin, by dropping all consonants that begin a word, and planting at the end of each word the original first consonant, followed by "eh". Here, in addition to making the request, the user is also describing to the system exactly how to implement the request. Later, they may ask their agent to turn the text into pig-Latin, without supplying the knowledge of how to do it.

Some manipulations occur automatically. For example, the user may have certain preferences about fonts. Once the agent has received the document, it may alter the nGen models to reflect these types of preferences. Many of these actions will result in across-the-board alterations to the nGen model. But even so, these are accomplished via the same recursive process of implementation.

nGen Sites

As noted earlier, there is no real distinction between an nGen site and a browser. One means of sharing content is for the user’s agent to send a proxy agent to the other system’s environment as described earlier. An alternative, which more accurately parallels "browsing," is for one agent to ask another for a copy of an nGen model structure, with relevant knowledge (this is akin to the browser asking for a page, or a set of pages, from the site’s server).

Once copied and sent, the receiving agent has full control over the nGen model structure. The user may, through natural language, interact with it, manipulate it, move parts into areas of their own environment for permanent storage.

This process of transferring nGen models, with associated knowledge, corresponds to many broad areas of conventional computing. The act of creating the site is similar to the use of platform independent languages, and conventional content creation tools like hypertext editors. The act of receiving, viewing, and tailoring the received nGen model roughly parallels a virtual operating system, and the browsers rendering of a document.

For example, the user is visiting, through his proxy, a virtual department store. Rather than interact with other customers, the user prefers to have the store copied into his own machine. The sites’ agent sends the user’s agent the relevant nGen models, together with all of their individual applied knowledge, a copy of the store’s agent, and any relevant applied knowledge that the site’s agent deems appropriate.

This copy of the store is navigable and viewable without any alteration. If however, the user wants to alter it they may do so. They may ask that an aisle be created consisting solely of digital cameras. Through interaction with the nGen models, and through the associated applied knowledge, the user’s agent can determine where all of the digital cameras are, and create an aisle consisting solely of them. The user may ask for complex alterations, such as "make the layout of the store resemble the store I just went to", in which case the user’s agent may have to draw off of its own applied knowledge and analytical tools, and perhaps interact with other agents elsewhere on the Internet.

The concepts described above are applicable to numerous areas of software development. For example, translation is easily applicable to programming languages, 4th generation languages, specification languages, incremental design language, non-programming language, command languages natural query languages formal query languages data entry languages, translation to and from different kinds of languages, natural languages, artificial intelligence, formal programming languages, executable code, byte code, and pattern recognition.

The operating system concepts are applicable to command languages, GUI, application interface, underlying paradigms, searching mechanisms, information storage, and file management.

The network infrastructure is applicable to the underlying mechanisms of transfer, search mechanisms, application transfer and execution, common GUI, command languages.

The Internet/intranet concepts are applicable to dynamic pages, 3D modeling, site construction, site-to-site communication, browser-to-browser communication, site-to-browser communication, personal communication, information storage, navigation, file transfer, and back-end application infrastructure.

While the invention has been described in detail and with reference to specific embodiments thereof, it will be apparent to those skilled in the art that various changes and modifications can be made therein without departing from the spirit and scope thereof. Thus, it is intended that the present invention cover the modifications and variations of this invention provided they come within the scope of the appended claims and their equivalents.
What is claimed is:
1. An operating system comprising:
   means for interacting with a user;
   means for storing information, the information being organized using nuclear trees;
   means for interacting with underlying hardware; and
   means for initiating an application to run on the underlying hardware.
2. The operating system of claim 1, wherein the operating system is a virtual operating system.
3. The operating system of claim 1, wherein the operating system is a continuously running operating system.
4. The operating system of claim 1, wherein the operating system is used for a multi-machine network.
5. The operating system of claim 1, wherein the nuclear trees include source trees.
6. The operating system of claim 1, wherein the nuclear trees include applied knowledge.
7. The operating system of claim 1, wherein the nuclear trees are used to interpret user input through experimentation.
8. An operating system comprising:
   means for interacting with a user using nuclear trees;
   means for storing information;
   means for interacting with underlying hardware; and
   means for initiating an application to run on the underlying hardware.
9. The operating system of claim 8, wherein the operating system is a virtual operating system.
10. The operating system of claim 8, wherein the operating system is a continuously running operating system.
11. The operating system of claim 8, wherein the operating system is used for a multi-machine network.
12. The operating system of claim 8, wherein the nuclear trees include source trees.
13. The operating system of claim 8, wherein the nuclear trees include applied knowledge.
14. The operating system of claim 8, wherein the nuclear trees are used to interpret user input through experimentation.
15. An operating system comprising:
   means for interacting with a user;
   means for storing information;
   means for interacting with underlying hardware; and
   means for initiating an application to run on the underlying hardware,
   wherein nuclear trees are used to control internal behavior of the application.
16. The operating system of claim 15, wherein the operating system is a virtual operating system.
17. The operating system of claim 15, wherein the operating system is a continuously running operating system.
18. The operating system of claim 15, wherein the operating system is used for a multi-machine network.
19. The operating system of claim 15, wherein the nuclear trees include source trees.
20. The operating system of claim 15, wherein the nuclear trees include applied knowledge.
21. The operating system of claim 15, wherein the nuclear trees are used to interpret user input through experimentation.
22. An operating system comprising:
   means for interacting with a user using a Graphical User Interface;
   means for storing information;
   means for interacting with underlying hardware; and
   means for initiating applications to run on the underlying hardware,
   wherein the Graphical User Interface uses nuclear trees to integrate outputs of the applications.
23. The operating system of claim 22, wherein Graphical User Interface also uses nuclear trees to integrate internal functions of the applications.
24. The operating system of claim 22, wherein the operating system is a virtual operating system.
25. The operating system of claim 22, wherein the operating system is a continuously running operating system.
26. The operating system of claim 22, wherein the operating system is used for a multi-machine network.
27. The operating system of claim 22, wherein the nuclear trees include source trees.
28. The operating system of claim 22, wherein the nuclear trees include applied knowledge.
29. The operating system of claim 22, wherein the nuclear trees are used to interpret user input through experimentation.
30. An operating system comprising:
   means for interacting with a user;
   means for storing information as applied knowledge;
   means for interacting with underlying hardware; and
   means for initiating an application to run on the underlying hardware.
31. The operating system of claim 30, wherein the operating system is a virtual operating system.
32. The operating system of claim 30, wherein the operating system is a continuously running operating system.
33. The operating system of claim 30, wherein the operating system is used for a multi-machine network.
34. The operating system of claim 30, wherein the nuclear trees include source trees.
35. The operating system of claim 30, wherein the nuclear trees include applied knowledge.
36. The operating system of claim 30, wherein the nuclear trees are used to interpret user input through experimentation.
37. An operating system comprising:
   means for interacting with a user;
   means for storing information;
   means for interacting with underlying hardware; and
   means for initiating an application to run on the underlying hardware,
   wherein nuclear trees are used to provide an infinitely expressive language.
38. The operating system of claim 37, wherein the operating system is a virtual operating system.
39. The operating system of claim 37, wherein the operating system is a continuously running operating system.
40. The operating system of claim 37, wherein the operating system is used for a multi-machine network.
41. The operating system of claim 37, wherein the nuclear trees include source trees.
42. The operating system of claim 37, wherein the nuclear trees include applied knowledge.
43. The operating system of claim 37, wherein the nuclear trees are used to interpret user input through experimentation.

44. An operating system comprising:
   means for interacting with a user using a Graphical User Interface and a general purpose translator;
   means for storing information;
   means for interacting with underlying hardware; and
   means for initiating applications to run on the underlying hardware,
wherein the operating system permits different applications to communicate to each other using source trees.
45. The operating system of claim 44, wherein the nuclear trees include source trees.
46. The operating system of claim 44, wherein the nuclear trees include applied knowledge.
47. The operating system of claim 44, wherein the nuclear trees are used to interpret user input through experimentation.
48. A hierarchical structure for storing information embodied in a computer readable medium comprising:
   a source tree for expressing a task; and
   applied knowledge to refer to at least one of an executable component and an interpretive component.
49. The structure of claim 48, wherein the source and the applied knowledge form a nuclear tree.
50. The structure of claim 48, wherein a nuclear tree is formed from the source tree and the applied knowledge.
51. The structure of claim 48, wherein the nuclear trees are used to interpret user input through experimentation.
52. A method of interpreting input from a user comprising the steps of:
   identifying a list of potential candidates to understand the input;
   recursively launching each candidate with the input; and
   evaluating results returned by each candidate.
53. The method of claim 52, wherein the candidates are launched sequentially.
54. The method of claim 52, wherein the candidates are launched in parallel.
55. The method of claim 52, wherein the input includes text.
56. The method of claim 52, wherein the input includes speech.
57. The method of claim 52, wherein the input includes an image.
58. A method of creating a document comprising the steps of:
   parsing content into a hierarchy of components, each component having its own associated knowledge base;
   collecting the associated knowledge base about at least one subset of the components into a set of source trees; and
   creating nGen models with nuclear trees that use the collected associated knowledge base and the set of source trees, which, together with the hierarchy of components, comprise the document.
59. The method of claim 58, wherein the components are two-dimensional components.
60. The method of claim 58, wherein the components are three-dimensional components.
61. The method of claim 58, wherein the remaining knowledge base not associated with the subset of components is associated with the whole document.
62. A computer program product for operating a computer comprising:
   a computer usable medium having computer readable program code means embodied in the computer usable medium for causing an application program to execute on a computer system, the computer readable program code means comprising:
   computer readable program code means for interacting with a user;
   computer readable program code means for storing information, the information being organized using nuclear trees;
   computer readable program code means for interacting with underlying hardware; and
   computer readable program code means for initiating an application to run on the underlying hardware.
63. A computer program product for operating a computer comprising:
   a computer usable medium having computer readable program code means embodied in the computer usable medium for causing an application program to execute on a computer system, the computer readable program code means comprising:
   computer readable program code means for interacting with a user using nuclear trees;
   computer readable program code means for interacting with underlying hardware; and
   computer readable program code means for initiating an application to run on the underlying hardware.
64. A computer program product for operating a computer comprising:
   a computer usable medium having computer readable program code means embodied in the computer usable medium for causing an application program to execute on a computer system, the computer readable program code means comprising:
   computer readable program code means for interacting with a user;
   computer readable program code means for storing information;
computer readable program code means for interacting with underlying hardware; and
computer readable program code means for initiating an application to run on the underlying hardware,
wherein nuclear trees are used to control internal behavior of the application.

65. A computer program product for operating a computer comprising:

a computer usable medium having computer readable program code means embodied in the computer usable medium for causing an application program to execute on a computer system, the computer readable program code means comprising:

computer readable program code means for interacting with a user using a Graphical User Interface;
computer readable program code means for storing information;
computer readable program code means for interacting with underlying hardware; and
computer readable program code means for initiating applications to run on the underlying hardware,
wherein the Graphical User Interface uses nuclear trees to integrate outputs of the applications.

66. A computer program product for operating a computer comprising:

a computer usable medium having computer readable program code means embodied in the computer usable medium for causing an application program to execute on a computer system, the computer readable program code means comprising:

computer readable program code means for interacting with a user;
computer readable program code means for storing information as applied knowledges;
computer readable program code means for interacting with underlying hardware; and
computer readable program code means for initiating an application to run on the underlying hardware.