Linköping Electronic Articles in
Computer and Information Science
Vol. 6(2001): nr 22

On the Design of Software Individuals

Erik Sandewall

Linköping University Electronic Press
Linköping, Sweden

http://www.ep.liu.se/ea/cis/2001/022/

Published on August 30, 2001 by
Linköping University Electronic Press
581 83 Linköping, Sweden

Linköping Electronic Articles in
Computer and Information Science
ISSN 1401-9841
Series editor: Erik Sandewall

Typeset by the author using LaTeX
Formatted using étendu *style*

Recommended citation:
<Author>. <Title>. Linköping Electronic Articles in Computer and Information Science, Vol. 6(2001): nr 22. http://www.ep.liu.se/ea/cis/2001/022/. *August 30, 2001.*

This URL will also contain a link to the author's home page.

Abstract

In this article we address the question of design principles for software individuals, and approach it as a software design issue. We use the term 'software individuals' to designate aggregates of programs and data that have the following properties:

- They exist in a population of similar, but not identical individuals.
- Individuals are able to interact with their surrounding environment, with each other, and/or with people. While doing so they may modify their internal state.
- Each individual contains the safeguards that may be required in order to select which influences to accomodate and which ones to ignore.
- The aggregate of programs and data that define an individual, and that in particular define its behavior, is a part of its internal state and can therefore be modified as the result of the interactions where the individual is involved.
- Individuals or small groups of individuals are able to create new individuals that inherit the features of their parent(s).
- The program/data aggregate that defines an individual is symbolic in character. The ability for knowledge representation is designed into individuals from the start.

Author's address

Department of Computer and Information Science
Linköping University
Linköping, Sweden

E-mail: `erisa@ida.liu.se`
Webpage: `http://www.ida.liu.se/~erisa/`

1 Introduction

We use the term 'software individuals' to designate aggregates of programs and data that have the following properties:

- They exist in a population of similar, but not identical individuals.
- Individuals are able to interact with their surrounding environment, with each other, and/or with people. While doing so they may modify their internal state.
- Each individual contains the safeguards that may be required in order to select which influences on its state are to be received and accomodated, and which ones are to be ignored.
- The aggregate of programs and data that define an individual, and that in particular define its behavior, is a part of its internal state and can therefore be modified as the result of the interactions where the individual is involved.
- Individuals or groups of individuals are able to create new individuals that inherit the features of their parent(s).
- The program/data aggregate that defines an individual is symbolic in character. The ability for knowledge representation is designed into individuals from the start.

The last item distinguishes software individuals from the software that is evolved by genetic programming.

In this article we address the question of design principles for software individuals, and approach it as a software design issue. This problem is an unconventional one from the point of view of ordinary software engineering. The emphasis on software self-modification runs counter to conventional software design principles, where the inherent ability of the von Neumann machine to modify its own programs is strongly restricted in particular through the design of common programming languages. In our approach, program self-modification is viewed as an important possibility, and not as an abberration.

Our proposed Software Individual Architecture is characterized by the following principles:

- The use of a Software Individual Kernel (SIK) which is a minimal individual having the basic properties of replication, self-modification, interaction, and ability to administrate and to extend its own state.
- The use of a Knowledge Object Repository that is accessible by, but external to software individuals, and that can be shared by several individuals in a population. This repository is also the basis for several of the user services that are built using the architecture.
- Each Software Individual Kernel is set up as a directory structure (directory and all its sub-directories) in the file system of a particular computer host. A population may 'live' within one host, or in a network of several hosts. Each individual is self-contained in this structure and each carries with itself (i.e., contains) a full set of the required programs and data defining it.

- An individual can acquire additional features, besides those present in its Kernel, by adopting additional program and data modules, which either are sent to it by other individuals, or by importing them from the Repository. A number of previously developed knowledge management services have been ported so that they can be acquired by the new Software Individual Kernel.
- We have worked the Software Individual Kernel so as to achieve maximal simplicity and generality.

The present paper describes the design of the Software Individual Kernel and the process that led up to it. It also discusses how the design and use of this kind of software differs from the design of conventional software. It furthermore discusses the relevance of Software Individuals as the basis for implementing machine intelligence systems.

2 The need for individuals

If you think of a machine intelligence as a system that is able to acquire knowledge, to communicate, and to learn, then it follows at once that the system must be able to persist for a considerable period of time. Rather than going straight from the design table to a demo or into production, the system will have to be worked with and allowed to acquire information from the real world or from interaction with others.

A researcher or designer that develops such a system is likely to try many alternative solutions to various aspects of the design. If it is built by a team, and not by a single person, then different team members are going to make changes concurrently to different parts of the design at hand. Contemporary software engineering systems for version management are intended for such situations, but in the particular case of developing machine intelligence systems another approach is possible, namely, that the designer or design team maintains a population of similar software individuals, causing changes in these individuals or causing them to acquire their own modifications and extensions.

From a machine intelligence point of view, this is a very natural thing to do: if we are developing a system that is going to act as an individual when it does its knowledge acquisition and while it is in use, then why not treat it as an individual already during the early design stage, and then onwards? From a software engineering point of view, the idea can be formulated as follows: instead of having a centralized version management system that keeps track of various configurations that can be obtained from existing modules and their various versions, why not decentralize this function in such a way that each active configuration (aka software individual) is aware of its own composition and is able to take responsibility for changes to it?

We are presently pursuing an exploratory research effort in order to try out some ideas and designs along the lines of what has just been described. Our approach is one of *meticulous programming*, which means that we focus on a small 'program' - the kernel system for software individuals - and work intensively on it in order to obtain a design that is as clean, simple, and powerful as possible. The relevance of the results is checked in concurrent activities where existing, relatively large application software is modified so that it can operate on top of the meticulously designed kernel.

This project is still ongoing and the results that have been obtained so far are preliminary, but quite promising. In brief, it has been possible to write a software kernel with about 600 lines of Lisp code that seems to have the fundamentally important properties for being used as the basis for software individuals. The program size is crucial here: if individuals are going to self-modify (by the initiative of themselves, their peers, or human operators) then the design must be as simple and coherent as possible.

We shall use the term *Software Individuals Architecture*, or SIA for the overall structure of software individuals, and the term *Software Individual Kernel*, or SIK for the small system that is the starting-point of every individual that is designed according to the architecture.

3 Technical preliminaries

Before proceeding to high-level issues such as the strategies for reproduction and for state extension in SIA software individuals, we must first explain the basic programming technology. In the SIA architecture each software individual is an aggregate of procedures and data; it is not only a program. Each individual contains its own copies of the programs that define its behavior. The program consists of many data-driven procedures, that is, procedures that are attached as properties to data objects.

The individual operates in interpretive mode. This is more or less a necessity if self-modification is going to be achieved.

In concrete implementation terms, each software individual is a directory structure from a certain root directory and down, including all its sub-directories. Different individuals are disjoint directory structures. Some files in the individual's directory structure contain data (as Lisp S-expressions or in an XML-like format); others contain Lisp function definitions; others again contain expressions in specialized embedded languages whose interpreter is also a part of the agent.

Since agents are relatively similar, and in some cases their directory structures and associated filenames are isomorphic, it follows that files with the same name often occur in corresponding positions of the structure of many of the individuals. The usual search-path concept in operating systems is therefore not very useful here: it is rarely meaningful to ask for the location, in the directory structure, of a file with a given name, since there will be one answer for each individual in the system. We therefore consistently use the convention that if one entity (typically, a file) in an individual refers to another entity, then it does so by specifying the entire path for the latter in the directory system.

This has the consequence again that moving an individual to another place in the directory system is not entirely trivial - there is always going to be a number of places in the individual's programs and data that depend on location, and that need to be modified when the individual is moved. The same holds of course if an individual or a kernel part of it is copied, which is what happens during reproduction, or if a copy of a part of one individual is acquired by another individual, which typically is what happens during 'education' where elder individuals contribute knowledge to junior ones. The need for such updates can be reduced by relative-naming constructs, but the experience has been that they can not be eliminated entirely. In particular, the remaining need for absolute naming occurs in the very kernel of the system, which is at the focus of our design interest.

3.1 Basic terminology

Finally we need to define how we use a few technical terms that work differently in an interpretive language, such as Lisp, compared to a compiling-oriented language. Precision with respect to terms such as "program" and "definition" becomes particularly important when we deal with self-modifying software. The casual reader may safely skip the present subsection, however.

The term *function* is used to include both "functions" and "procedures" in other languages. A *function assignment* assigns a *function definition* to a *function name*, and is expressed as a piece of text.

When we talk about a *program*, we shall mean one text file that contains a sequence of assignments, including both function assignments, assignments of values to constants, and assignments of properties for objects (e.g. on property-lists). Such a program is not in itself said to be "run"; it is merely *loaded* which means that the function definitions and the associations from function name to function definition are re-represented as data structures in the current Lisp session. - Programs can be correctly referred to as files, since they are a particular kind of files. Notice in particular that the SIK is not in itself a program, with this terminology: it *is* a software system containing a number of programs.

A *request* (our term) is a textual expression combining a function name and a sequence of arguments expected by it. Function assignments are requests, in particular, using a function-definition-assigning function. The contents of a program file (= program) is in fact always a sequence of requests.

A *listener* is a function that receives successive requests, usually as typed in by the user, and that executes the corresponding function definitions using current bindings in the session at hand. The Lisp system contains a standard listener (the so-called read-eval-print loop), but it is possible to replace it by another listener using a request.

The following is the normal execution structure when the SIK software is used. One *run* (this term will be put in its proper context later on in this article) is initiated by giving a *program* to the *Lisp system*; the latter starts up and *loads* the program. Some of the requests in that program ask to *load* other *programs*, often recursively but not to any great depth. As a result, the session is filled with bindings from *function names* to *function definitions*, as well as with similar bindings for constants etc.

In many cases, the last request in the last file to be loaded replaces the standard *listener* with another one which has been defined earlier on in the loading process. In any case, the current listener obtains control when the program loading process is completed, and the subsequent computation depends on the input from users, pipes, or other sources as determined by the listener. The combination of the loading process and the subsequent listener-controlled computation is referred to as the *run*, and also as the *execution* of the top-level program where the loading process started.

Notice, in particular, the distinction that is made here between *loading* and *executing* a *program*. Notice also that requests occur and assignments are made even when the program is loaded. In fact, the Lisp system will allow almost any computation to be invoked from a program file that is being loaded. The SIA architecture does impose some restrictions as to what is allowed at load-time, in order to facilitate the realization of replication and self-modification, but even so the loading process retains a considerable degree of flexibility. The operation of advising, which will be explained later on in the article, is one example of an operation that usually takes place when programs are loaded.

4 Reproduction and education strategies

Reproduction is a necessity in biological systems since organisms may be eaten by other organisms, and since they deteriorate physically with old age and need to be replaced. Neither of these reasons applies a priori for software, except that a long sequence of updates leads to deterioration in many software systems. However, the use of multiple individuals in the design and development process anyway raises the question how the different members of a population are to be generated. Should one always start with the same initial system, or should one make additional copies of existing systems at their current state of development, or is there another method?

Genetic programming uses the method of randomized mutation and evolutionary selection, but in our case we need a method that can be controlled more effectively. We propose to use a reproduction cycle that consists of the following steps:

1. Obtaining a copy of a Kernel Individual (KI)
2. Programmed Acquisition of structures in the KI, in order to build a fully developed individual
3. Self-Test
4. Evaluation for acceptance in the community
5. Reproduction, generating additional copies of the KI

This cycle will be referred to as KIPASTER, forming an acronym from some of the keywords. Each individual has originally been obtained by starting with a Kernel Individual that has the basic facilities, and extending it or getting it to extend itself by acquiring more programs and data, together referred to as structures. It is able to initiate new individuals that are not copies of itself at present, but which are only copies of its own kernel. Each new kernel individual is generated with an *acquisition program*, that is, a script for what additional structures it is to acquire at the beginning of its existence. Acquisition may consist of obtaining plain copies of structures in other individuals (the initiator, or others), or it may involve sending questions to other individuals, asking them to prepare presentations of some aspects of themselves, or again it may consist of taking this information from other individuals and modifying it before it is incorporated into the overall structure of the acquiring agent.

Structure acquisition per se is an inherently unreliable process, since it is difficult to guarantee that the contributions from other individuals will be compatible with what is already in the receiving one. It can be made reliable using additional software support. The KIPASTER paradigm therefore includes a Self-Test step, where the new individual makes various checks on its own functioning. At the end of the self-test the individual reports to the community where it was created that it aspires to become a member there. This naturally allows the community to also evaluate the individual in order to assess its qualifications, before it is adopted. After this, the new individual is able to both participate in continued interactions with other members of the community, and to initiate its own descendants.

The KIPASTER paradigm can be used both under human operator control, and as an autonomous reproduction process. A human operator may control it tightly by always deciding on the creation of new individuals, and on the structure acquisition script for each of them. In this case the process supports the operator through the self-test and evaluation facilities, which can be done automatically under program control. Alternatively, the KIPASTER process may be entirely autonomous to the community, if the initiator decides on the creation of the new kernel individual and its acquisition program, possibly after interaction with other individuals, and the community is equipped to make the evaluation and acceptance decision based on overall criteria.

The KIPASTER paradigm does not provide any counterpart of the bisexual reproduction scheme that dominates in plants and animals. Adopting such a scheme is not a goal in itself, and we see no particular advantage with it for the kind of symbolic-oriented individual that is considered here. The use of a single initiator means that each individual has a well-defined structure for its kernel, which facilitates the structure acquisition process.

5 Methodology: grooming of a set of software individuals

The design of the software individual kernel (the SIK) is very important for the KIPASTER paradigm. The SIK must have all the functionality that is needed for reproduction and for the acquisition of structures. At the same time, it should not contain any unnecessary facilities, since subsequent changes in the design of the SIK are by definition going to be much more difficult than changes in those structures that can be acquired by a new SIK. For the same reason the design of the SIK should be as simple and transparent as possible.

We have therefore chosen a research methodology where we initially perform manually those functions in the reproduction process that shall eventually be done automatically. We started with an initial implementation of a SIK along the lines just described above, where in particular each individual was able to initiate new individuals. We also set up several *populations*, each consisting of several individuals, and used them for further extension of the facilities. Different populations were placed on different computers, although each particular population has all its member individuals on the same computer. We then started modifying the kernel in order to incorporate additional important features; we also started building a library of structures that a SIK could incorporate.

In this way we intentionally created a situation where software updates in one individual had to be communicated to the other existing individuals, and we tried to arrange things so that this transfer of software impulses could be automated instead of us having to make the same modifications on several individuals. Just making a sufficient number of copies of a newly changed individual and replacing the other individuals with them was ruled out as cheating: the exercise required (in principle, at least) that the newly changed individual *communicated* its changes to the others.

In fact, the option of cheating by copying soon became irrelevant anyway, since individuals started to differ in a number of interesting ways. Each population needed one designated individual that kept track of the others. Also, different populations were set up to use different operating systems (Unix, Win9X, etc) and different Lisp interpreters (Allegro Common Lisp, Xlisp) that are not entirely compatible. Sometimes, different individuals in the same populations have used different Lisps.

Ideally the individuals should have the ability right from the start to communicate their updates to each other. In practice this was not the case, so we did spend some time making the same updates manually in several individuals. However, the obvious meaninglessness of this provided a strong incentive to find a solution that eliminated the chore.

The development of the SIK was not done *in vacuo*; from time to time in this process we have also programmed the new individuals to incorporate additional structures that were taken from earlier software developments. This was done in order to insure that the SIK design is indeed capable of incorporating structures and of transmitting the information which of them to incorporate, and how to do it. The modules that are being incorporated in this way are for document preparation support, journal management, and simulation of vehicle movements.

One other aspect of this methodology is worth mentioning: each population of software individuals has its own *history file*, that is, a text file where we have made careful notes of all changes that were made to the population, and to the individuals in it. This practice serves several purposes: it allows us to go back and check what has happened, and to keep track of the rationale for various design choices. It also provides a body of experience for the future when we look again at what the development process is like for software individuals and how it should be supported. Finally, the mere fact that one has to take notes as one works with a population of software individuals gives greater assurance that every change is well considered and not made in a haphazard manner.

The concept of careful control of software changes is a standard one in industrialized software development, but it is not a standard technique in the context of exploratory software development like in the present project. The experience with this technique has been very positive.

6 The emerging architecture for the SIK

The methodology for exploratory software development that has now been described, has so far resulted in the following design.

6.1 The individual

Each individual consists of a set of *processes*, a set of *materiel files*, a set of *resource files*, and a set of *pipes* for communication between processes. All of these are primarily represented as structures of directories and files, although processes and pipes also have a dynamic aspect.

6.2 Processes

The use of several processes allows the individual to dedicate one process, the *progenitor*, to the task of importing external structures. Therefore the kernel individual (the SIK) contains only the progenitor process, and all other processes are obtained as a result of extension.

Additional processes in the same individual may be used for productive purposes, performing what the individual is supposed to do as a service. Yet other processes may serve internal needs, such as for optimization of the individual's behavior, for learning from experience, or for checking incoming information and proposed updates and thereby to protect the integrity of the software individual. It seems quite natural to separate these functions as distinct processes, so that they can be acquired, evolved, and operated relatively independently of each other.

We use the term process in a slightly different sense than what is usual in computing systems. For us, a process is primarily a substructure of the individual - a tree consisting of a particular directory and its subdirectories. The process directory is in fact a non-immediate subdirectory of the directory representing the individual.

A process also has a dynamic aspect in terms of computations, which are organized as a sequence of *runs* in the sense that was defined in section 3. Each run is obtained by executing a program (in our technical sense) during a limited period of time, and it has the effect of changing the *stored state* of the process from an *initial state* at the beginning of the run, to an *ending state* at the end of the run. At most one run can be executing at any point in time, in a given process. The stored state of the process remains unchanged during intervals of time where no run is executing, except in those exceptional situations where a process directly changes the stored state of a neighboring process.

Runs are in one of several possible *modes*, where each mode is characterized by what programs are loaded into the run and what listener is used. For example, the *commander* run-mode is defined as a read-execute-print loop where successive requests of a particular kind (kernel commands) are obtained from the user and executed in the process. The *server* runmode is defined to receive kernel commands from one or more pipes that connect processes in the same individual, executing them in the order of arrival. The *cgi* runmode executes one request that is sent to it from a web browser, and so on.

The stored state of a process is organized as a set of *buckets* each of which contains assignments to state variables (global variables in the Lisp system), to properties on the property-lists of symbols, and to other similar slots in the hosting Lisp system. Buckets are organized like files in Interlisp, which means that each bucket has a *table of contents* that specifies which state variables and properties are assigned a value in the bucket. During a run it is possible (using appropriate requests) to load a bucket (meaning to do a Lisp “load” operation on the bucket file), to change the values of state variables and properties that are defined in the bucket, and to re-generate the bucket file based on the current values of the variables and properties in its table of contents.

Therefore, the static structure of a process consists of a set of named buckets, a set of named runmode definitions[1], and sets of incoming and outgoing pipes for the process. Runs in the process are initiated by a user, by other processes in the same individual, or by a separate scheduler. Runs are terminated by their own decision, or by an intervention by a user.

6.3 Motivation for the two-level process structure

To summarize, the evolution of a SIK process proceeds on two levels. There is the *system level* where the process performs a sequence of runs, each of which transforms a stored state for the process to the next stored state. On the more detailed *dynamic level* each run is a computational process that is overseen by the operating system in the conventional way.

The philosophical reason for this division is that it provides checkpoints where it is practical to do self-modification of the programs, or more precisely, of the aggregate of programs and data that together constitute the process. In addition there are a number of convenience reasons. The very concept of a software individual requires that it should be able to persist over a considerable period of time, amounting at least to months and hopefully to years. Conventional desktop computers and operating systems are not designed with such longevity in mind. Other systems (such as those being used in telecommunication systems) have the robustness that would be required, but they might not be appropriate for the present purpose for other reasons.

The structure of the process computations as a sequence of runs allows us to work with an assortment of different runmodes, such as the commander runmode where a user can issue a sequence of commands interactively, and a server runmode. This is also a way of obtaining modularity, since if we were to dispense with the two-level structure and work with one single computational process that persists for the entire lifetime of the individual, then that process would have to implement all the runmodes within itself, which would likely be a less modular design.

6.4 Resources and materiel

Besides the processes, an individual also contains *resource files* and *materiel files*, which can be used during process runs in all the processes of the individual.

Between runs, the state of a process consists only of its stored state, that is, its bucket files. Within a run, however, there is also the *dynamic state* of the computation, which is preserved and modified between successive executions of requests by the listener. In the normal way, the dynamic state consists of

[1] To be precise, runmodes are a special kind of buckets that are associated with a few extra, autogenerated files

associations from function names to function definitions, and similarly for constant variables, properties, etc.

A *materiel file* is a program that changes the dynamic state of the run where it is loading. It is not supposed to have any other effects. Each runmode has its own selection of materiel files that are loaded in the initial, loading phase of every run in that runmode, together with buckets that are also used in the loading process.

A *resource file* is a program that changes the stored state of its own process when it is loaded during the execution phase (rarely the initial loading phase) of a run. In special circumstances it may also change the stored state of neighbor processes within the same individual. A resource file is therefore used as an installation program: each facility that is available for being incorporated into individuals is typically represented by one resource file that 'installs' the facility and one (or more) materiel files that are loaded by runs once the facility has been installed.

The structure provided by resource and materiel files is a key element in the PA (programmed acquisition) part of the KIPASTER paradigm, since it endows individuals with a simple and clean mechanism for extending their own contents.

6.5 Kernel commands

The SIK implementation defines a range of *kernel commands* for operations relating to the use of buckets, pipes, materiel and resource files, as well as the import and export of the latter two structures. The implementation of commands is based on the architecture of the SIK since commands are able to change most aspects of the architectural structure. At the same time, commands are used in several ways within the architecture: they can be entered by the user in the commander runmode, they can be sent along pipes, they can be used in resource files, and so on. The kernel command facility is therefore an integrating part of the SIA architecture.

Technically speaking, commands constitute an alternate set of functions that only work with their own listeners.

Definitions for most of the commands in the SIK kernel are found in the materiel driver file. Other, optional materiel files may add further command definitions.

6.6 Advising

Although many extensions of functionality can be obtained by adding more kernel commands to the reportoire at hand, there are also many facilities that require amendments to existing commands. In order to achieve good modularity it is very important to have a way of serving this need without having to explicitly modify the definition of the original command in the materiel file where it happens to be located. We obtain this by using a technique that was developed in Interlisp a long time ago, namely *advising* [Teitelman, IJCAI 1969, p. 1]. However, advising in Interlisp could be applied to virtually any point in the definition of any Lisp function, which effectively restricted it to an operation performed by human users - a kind of high-level editing operation. In our case, we restrict it to advise given to kernel commands. Thus if one materiel file contains a definition of a kernel command C, then another materiel file may contain an invocation stating that whenever the command C is executed, a particular additional operation is to be performed before, or after the main operation. In fact, the materiel file containing the advise can even be loaded before the file containing the main definition.

6.7 Initial configuration and bootstrap of an individual

We are now ready to summarize the structure of the Software Individuals Kernel (SIK) which is the initial configuration of a new individual. It consists of the following parts:

- One process, the *progenitor*, containing one runmode, the *commander*, and one bucket dedicated to the commander runmode
- Four materiel files, namely:
  - The *bucket loader*, containing a few function assignments that are needed in order to load a bucket file;
  - The *driver*, containing basic function and command assignments that can be used in most of the runmodes;
  - The *environment specific definitions*, which accomodate the differences between different operating systems and Lisp systems;
  - The *commander definitions* which defines and activates the listener that is used in the *commander* runmode.
- One resource file, namely the acquisition program for the new individuial.

When a new individual is created, a new structure is set up where these components are direct copies of the same components in the initiating individual, with a few small but important exceptions. First, some of the properties in the commander bucket are assigned new values that represent the new individual, its name, its location in the file structure, and its ancestry. These values are specific to the individual, and are inherited by all other processes when later they are spawned in the new individual. Secondly, the acquisition program is not necessarily an exact copy of the initiator's original acquisition program, since the initiator and the initiation process may choose to put a modified or alternative acquisition program in the new individual.

Once set up, the new kernel individual proceeds to start a run in a specialized *acquisition* runmode in its progenitor process, where it loads and executes the acquisition program. Being a resource, this program can extend the contents of the progenitor process, and spawn additional processes and add content to them. It can also fetch materiel and resource files from a senior individual or from the knowledge object repository (KOR) serving the population, and add them to the structure of the present individual.

There are several ways for a process to add content to the stored state of another process:

- It may send messages to the new process through a *pipe*. As a general rule, when a process A spawns another process B, it always creates a pipe from A to B. Process A may send update commands through that pipe, although it must also make sure that process B has a run in server runmode where it honors the arriving requests.

- The spawner may request a run in *exec* runmode in the spawnee. The *exec* runmode is similar to a cgi-script invocation in that each run only executes one command, and then it is done. Such invocation of *exec* runs are of course subject to the restriction that there can be only one run at a time in each process.
- Finally, the spawner process may directly modify the file structures and file contents of the spawnee. This 'chirurgical' approach is often the easiest one to implement, but it may also be the most brittle one.

We are experimenting with all three of these methods, and we do not yet have any well-founded opinion concerning when each of them is appropriate.

7 Implementation experience

The present effort is one of explorative software development: we start with an initial goal formulation and an initial software design that only partly satisfies the goal. The design is modified in order to achieve the goal more fully. During that process, and as we gain some experience of using the system being developed, it is also natural to revise the goal statement from time to time. The process terminates successfully at the point where the design at hand fully satisfies the goal at hand.

The methodology section above explained why this method was considered appropriate for the present purpose, and what steps we have taken in order to have a steady course in the work.

The effort has been structured into three major steps, namely, one initial step and two concurrent later steps:

1. Initial step: design a crisp software individuals kernel that has the capacity of replication, self-modification, communication between individuals, awareness of and administration of its own state, and acquisition of additional facilities in a systematic way.
2. Machine intelligence step: explorative use of the SIK that was developed in the initial step, for an attempt to set up, educate and maintain a population of software individuals that are able to receive software updates on a high conceptual level and to communicate them to peers, as well as to optimize their performance and to learn from experience.
3. Application step: use the results from the initial step for some practical applications. In that context, experiment with also using the individuals that were developed in the machine intelligence step for those applications.

7.1 The initial step

At present we can only report on the initial step. The goals in that step have a software engineering flavor, as follows:

- The *closure requirement* has been the most important one, namely, that the characteristic properties of the SIK should support each other completely. For example, it is not sufficient that the SIK as designed by the programmer has the capacity of acquiring additional facilities, but that capacity must also be there in other processes that have been spawned within the individual, and in other individuals when they are initiated. The inter-process communication mechanism using pipes must be able to transmit acquisition information. Modifications of the reproduction mechanism must be transmitted to offspring, and so on.
- The SIK design should be crisp, in the sense that it contains no unnecessary code, no duplication of designs (never two separate features or constructs where one would be sufficient), and designs that are as general and powerful as possible while retaining the basic simplicity. We consider it worthwhile to redesign the software repeatedly in order to achieve this goal, and we would not be satisfied with the first implementation only because it worked.
- Minimization of the need for operator intervention in the processes of reproduction and structure acquisition, and in particular of the need for repetitive operator intervention where similar interventions are made several times in succession.
- Platform differences (operating system and Lisp system) should be as isolated as possible, ideally to only one file that can be exchanged according to platform, and which was referred to above as the platform specific definitions.
- To the greatest possible extent, extensions to the system should be defined and implemented as separate modules that can be acquired by an individual in a modular fashion.
- On the other hand, in those cases where improvements must affect the kernel itself, incremental improvements should only lead to small and local changes in the program.

At the time of writing this article, we have a SIK that has evolved as the result of a number of modifications to the original design, and that comes close to satisfying these requirements. Many of the iterations in its development were due to the closure requirement; many others were due to the crispness requirement. The accomodation of platform differences was introduced at a fairly late stage in the process, and did not offer any particular difficulties.

The total size of the SIK is about 600 lines of code in the printouts of the files with normal spatiousness and full legibility. This count includes both programs and OS-related files (e.g. .bat files that are used for starting runs), but it does not include empty lines, comment lines, and lines only containing right parentheses. We are convinced that even with the remaining additions the count will stay well below 800 lines of code. This size of a program is of course easy to understand and to manage.

7.2 Self-modification and documentation

Before it has started its acquisition process, a newly generated kernel individual consists of twelve files using eight directories. Both of these numbers increase when the individual acquires additional structures. In order to understand how the implementation works, one must understand both the mechanics of the directory structure and the contents of the files contained in it. When the individual begins to operate, first by executing its acquisition program, and later on by incorporating additional information, then both the directory structure and the contents of the files change in a number of ways. Bucket files change when a process adds information to a bucket and regenerates the bucket file. The generation of new runmodes causes the generation of additional files (.bat files in the Windows implementation) that are instrumental for invoking runs in the new mode. Newly added or modified materiel files may contain *advise* requests that amend the functionality of procedures and commands whose original definitions are in earlier materiel files.

It would therefore be quite a bit misleading to talk about the implementation as a program in the ordinary sense of that word. That is also why we have recycled the term 'program' for our purpose. Of course the SIK structure is a program in the more general sense that it defines the behavior of the computer hosting it, but a very different kind of program. Self-modification, which is the most characteristic feature of the SIK, is also an inherent property of the von Neumann machine, but it is a property that later generations of programming language designers have successfully locked in so that it shall not be available to users.

Self-modification is available everywhere in a certain sense, anyway, since it is easy to write programs that change the contents of files in the operating system, including files that contain programs. Software installation software does exactly this. This is usually considered to be of negligible interest from a scientific point of view, however, and even textbooks on compilers and other aspects of program development tools treat it marginally if at all.

We are therefore treading relatively unknown territory when we address the question of how self-modifying software such as the SIK is to be documented. It is entirely clear that comments in the program listings are of margnial value here. In order to understand how the implementation works, one must begin with an explanation of the upper one of the two levels for processes: the structure of directories and files, and how it may change as the result of various operations by the user or otherwise. After this, one can proceed to looking at the contents of the program and data files, and understand how they cause the changes and also how they are affected by the changes as the system modifies itself. Exploring how such a system documentation shall best be organized is yet another aspect of our project.

8 Related work

The view of programming languages and systems has changed considerably in the artificial intelligence community. AI was widely perceived to have its own software base at least until the early 1980's, with the dominance of Lisp and other quasi-functional programming languages, the attempts to commercialize specialized Lisp machines, conference sections on "programming languages for AI", and so on. All of this now seems very remote as AI software is commonly written in C++ or Java. Lisp, although still in use in many quarters is barely mentioned. Prolog, which challenged Lisp as a programming language for AI is in a similar situation.

One may debate whether the language deserved this fate, whether its widespread reputation for inefficiency is correct, and whether this development was unavoidable. In the context of the present article, we just observe that the kind of design that has been described here would be very much more difficult to achieve in a conventional programming language. It employs three important features of Lisp: its interpretation-oriented character, its standardization of a textual representation for data structures so that these can be printed onto files and later read back, and finally the representation of programs as data structures.

It is therefore not surprising that the account of related work must be virtually empty: this type of research is simply not in vogue today. Fashions do change, however, and we believe that work on core software principles for machine intelligence will sooner or later attract the attention of the research community again.

In the historical perspective, one may also ask whether similar work was not done during earlier stages in history. So far we have not found any matching attempts. One cue to the possible reasons is that the architecture described here relies on a using the directory structure dynamically: even a moderately sized software individual contains several tens of directories ("folders" in Windows jargon) on different levels, and more are added as modules are acquired. However, the use of large numbers of files and directories for organizing information is a relatively recent practice, which is illustrated by the fact that even the official definition of Common Lisp does not specify any function for creating a new directory. More generally, we believe that the approach described here would not realistically have emerged from the point of view of the A.I. programming practices of earlier decades.

We therefore think of this project as an effort where we return to some of the roots of artificial intelligence research, combining a half-lost tradition with contemporary opportunities. After all, the design of a long-lived software individual with the ability to modify itself and to reproduce, ought to be at the very core of machine intelligence research.

There is also another perspective on the present work, where it can be seen as a novel and unorthodox approach to software engineering. That, however, is another story.

9 Acknowledgement

The research reported here is done in the WITAS project which is supported by the Wallenberg Foundation.