CASL

PM 2003:3

2003-12-10

The Laboratory for Cognitive Autonomous Systems
Department of Computer and Information Science
Linköping university, Sweden

Erik Sandewall

A Software Architecture for A.I. Systems based on Self-Modifying Software Individuals

Related information can be obtained via the following WWW pages:

Linköping university:	http://www.liu.se/
WITAS Project:	http://www.ida.liu.se/ext/witas/
Author's Department:	http://www.ida.liu.se/
The author:	http://www.ida.liu.se/~erisa/

This report is a slightly extended version of a paper that was presented at the 2003 International Lisp Conference in New York, USA, in October 2003. The extensions consist of the addition of subsection 3.2, plus some minor corrections and typographical improvements.

Abstract. *The Software Individuals Architecture (SIA) is a framework for defining software systems that are capable of self-modification and of reproduction on the level of an interpretive programming language. In abstract terms, a self-modifying system is a labelled tree containing* scripts *at some of its nodes; these scripts are effectively programs. A* computation *in such a system executes a specific script. In doing so it maintains a local computational state, but it also uses and updates the labelled tree. The labelled tree, the local computational state, and the command language used for the scripts are all designed in such a way as to support self-modification and reproduction in a structured and orderly fashion.*

We have defined a practical system of this kind both on an abstract and formal level and as an implementation using Lisp as the host language. This architecture has been used as a platform for several applications, including in particular the speech and natural-language dialogue system for an intelligent autonomous unmanned aerial vehicle (UAV) in the WITAS project. The architecture design has been revised repeatedly as a result of using it for this application as well as several others.

1 Self-Modifiying Software

1.1 The Case for Self-Modifying Software

We are interested in self-modifying software systems for the following reasons:

1. It is a very little studied topic. The idea of self-modifying software is met by many by disdain bordering on a taboo.
2. If you consider the entire set of software in a computer as *the* program of the computer, then all such global programs are self-modifying. For example, installation of new software on a computer constitutes self-modification of *the* program. Correct and robust software installation is an important and complex problem.
3. Several software plagues, including viruses, use self-modification in order to survive the countermeasures.
4. On a philosophical note, it is a remarkable fact that the von Neumann computer architecture is based on program-data equivalence, which means that machine-level programs are inherently capable of self-modification, and yet all conventional programming languages make a conscious design decision not to allow the programmer to modify the program. In other words, one of the most significant properties of the hardware is taken away from the programmer. It is interesting to explore what a programming language and system will be like if self-modification is allowed and supported properly.

These are reasons for considering self-modifying software as an important approach to software technology that offers new challenges and that has been very little studied.

1.2 The Case for Software Individuals

Besides the software-engineering motivation described above, our interest in these issues also has an artificial intelligence background, which leads us to impose a number of other requirements on the system, besides self-modification as such. The argument goes as follows: The long-range goal of AI is to build *systems that have and use intelligence of many kinds and over a broad spectrum*. Some of the necessary requirements on such a system is the ability to learn, and to acquire knowledge. This is only meaningful over longer periods of time; an AI system should have a lifelength of months, and preferably years. It should be able to modify its own characteristics during that time. However, ordinary computers do not easily sustain computations that go on for years, in particular not if the same system must be highly flexible and adaptive. Therefore we need a notion of software that can exist and change itself over long periods of time - a notion that combines concepts from programs and from databases. (The collection of data in a database has of course a high degree of persistence).

The present paper attempts to combine the software- technology aspect and the AI aspect of our work, but with an emphasis on the former.

1.3 Definitions

We begin with the definitions for a generic self-modification machine that is relatively abstract. Later on we shall specialize these generic concepts so as to become more concrete.

We define a *self-modification machine* as a fivetuple `(C, I, S, W, w0)`, where

```
C   is a language, the expressions in which
    are called *commands*
I   is an *interpreter* for expressions in C
S   is a domain of possible *system states* for
    a *self-modifying system*. Every such state
    is a structure containing labelled
    *elements* each of which is a sequence of
    commands in C
W   is a domain of possible *working states*
    during a computation with S
w0  is a designated member of W, the *initial
    state*
```

Whenever we refer to `C` below we mean the command language in our self-modification machine, and not the programming language C.

The interpreter is a mapping from `C x S x W` to `S x W`. In other words, given a pair (s, w) where s in `S` and w in `W`, each command c will produce a new pair $I(c, s, w) = (s', w')$. For convenience we write this also as $I(c, (s, w))$.

If $p = \langle c1, c2, ...cn \rangle$ is a sequence of commands, then we write $I(p, s, w)$ in the obvious way for $I(cn, ...I(c2, I(c1, (s, w)))...)$

We also write $J(p, s)$ for the first element of $I(p, s, w0)$. Informally, J is the operation of applying a program p to a system in state s, starting with the working state `w0`, and discarding the working-state part of the component when all of p has been executed.

If s is a system state and l is a label used in s, then we write *program*(s, l) for the sequence of commands that s associates with l. We extend the use of the operators I and J as follows:

```
I(l,s,w)  =  I(program(s,l), s, w)
J(l,s)  =  J(program(s,l),s)
```

The self-modification machine that is characterized by these definitions is such that the system state of the self-modifying system can contain both 'programs' and 'data', it is possible to invoke programs that are located within the system, and the result of executing such a program can change the state of the entire system, including the programs that it contains and the labels that those programs have.

2 The Abstract Software Individuals Architecture

The Software Individuals Architecture (SIA) is an experimental implementation of a self-modification machine in the sense defined above, with a particular goal of exploring the ideas mentioned in the AI introductory section. We wish to design a system that is capable of self-modification and reproduction on the level of the software language(s). Other approaches using genetic programming, for example, are not being tried in this project.

The system state in SIA is a labelled tree with a particular structure, called a *residence*. SIA residences contain subtrees called *software individuals*, and one of the key capabilities of a software individual is the ability for *reproduction*. That is, there are operations that are done relative to a particular individual in the residence, and whose effect is to construct one more individual and add it to the residence, as a neighbor of the breeding individual. The newbred is not in general a clone of the breeder.

Self-modification in SIA applies for individuals, therefore, and the changes that an individual can perform on the rest of the residence outside itself are fairly constrained. The most important operation is that an individual can *breed* another individual, but when doing so it only creates a simple individual with minimal contents. Then it is up to that new individual to acquire information from its breeder and from other neighbors, similar to how a child picks up knowledge from many people and not only from its parents. It is expected that this design will provide a high degree of robustness for the overall system. It is important to address the robustness problem since self-modification in itself introduces new dangers of software errors, while at the same time it also offers new ways of dealing with those errors.

2.1 The formal and the practical versions of SIA

The SIA residence can be understood in a formal way and in a concrete implementation sense. Formally, a SIA residence is a finite tree with the following characteristics:

- Each node in the tree has a label, which is a symbol, written as a conventional identifier (letters, digits, a few special characters allowed).
- The daughter nodes of a given node must have different labels. Therefore, each node in a tree can be characterized by the labels on the path leading to that node, including its own label.
- Besides daughter nodes, each node also has a set of labelled *scripts*. Each script is a sequence of commands in the command language `C`. The labels for scripts are of the form `a.x` or `a.f.x` where `a` is a symbol, `f` is chosen from a small set of *aspects*, and `x` is chosen from a small set of *extensions*. The scripts that are attached to a given node must have different labels.

A SIA label path therefore corresponds to a 'label' in the sense of the abstract definition in section 1.3.

In the concrete implementation sense, a SIA residence is understood as a directory structure (directory, subdirectories, and contents) in a computer system running a conventional operating system, such as Linux or Windows. Nodes in the formal interpretation of a residence correspond to directories; scripts correspond to alphanumeric files containing the sequence of commands in textual form; labels are the names for subdirectories and files. (Labels of the form `a.f.x` are implemented as `a.fx` using an extended set of extensions, in our current implementation). But in addition, the concrete implementation of the SIA architecture allows a greater variety of script languages, so that some scripts may contain programs in ordinary programming languages, and other scripts may contain data that are represented in particular formats.

Note, however, that when other types of operating systems, and in particular their file systems become available, then it will be possible to realize the tree structure of the abstract SIA architecture in entirely other ways than mapping it on the directory and file structure.

The formal interpretation of SIA architectures only models some aspects of the concrete implementation, therefore. At the same time, some choices in the formal definition of the architecture are motivated by considerations in the implementation, and will appear as ideosyncracies unless one is aware of that background. It is important to understand that the formal model of SIA has not been designed abstractly; instead it is an abstraction of an existing software system that has evolved with multiple applications over several years, and with successive redesigns in order to meet the needs of those applications. The applications included the speech and natural language dialogue system for an intelligent UAV in the WITAS project, and a support system for the editing process in scientific journals.

The following are some considerations that were important in the concrete implementation, and that are partly visible in the formal interpretation as well:

- A SIA residence shall resemble a mobile agent in the sense that it can easily move between hosts. For example, if a SIA residence is stored in a mountable memory device, such as a USB flash memory, then it shall be possible to mount that device into an arbitrary computer and immediately start to execute programs in the individuals of the residence. Similarly, if the residence is stored on a server then it shall be possible to run it on different hosts even in cases where the execution makes use of resources that are available locally on the hosts (that is, not on the server) and that may appear differently on different hosts.
- A SIA residence shall also be absolutely portable between different operating systems and programming languages, so that the tree representing the residence can be moved to another such environment and can perform computations there immediately and without cumbersome reinstallations.
- SIA-based applications use an extended command language with the following facilities in addition to those that are included in the generic model for self-modifying machines in Section I above:
  - providing parameter data to programs that are executed in an individual
  - input and output of data, for example through a web browser
  - invoking system software of various kinds (web servers, databases, etc)
  - allowing SIA scripts to contain programs in programming languages, e.g. in Lisp, and not only in the SIA command language.

Later sections will describe the present, concrete version of the SIA architecture. At this point we proceed with the formal version.

2.2 Understanding self-modifying software

As a result of working with self-modifying software during several years, we have made the following observation that we think is very important: *self-modifying software must be understood in very different ways from ordinary software.* In particular, ordinary programs are commonly thought of as 'documents', and it is often assumed that the ideal programming language would be one where programs can be read like their own documentation. This assumption does not work well for self-modifying programs, and trying to understand self-modifying programs like if they were ordinary ones will only lead to confusion.

The proper way of understanding self-modifying programs in general may still be a research issue, but for the SIA architecture we recommend thinking of the software as a collection of nodes and their associated scripts for which there are several distinct structures, namely:

- The quasi-static *residence structure* of the SIA residence as a labelled tree
- The *startup structure* which specifies how one script invokes another one during the startup of a computation in an individual within the residence

- The *reproduction structure* which describes how a residence may be extended with additional individuals, and how an individual may extend itself with additional components during a computation

We refer to the residence structure as quasi-static since it stays mostly put, but occasional changes occur, for example when a new individual is generated. It therefore changes much less often than the working state during a computation, and not at all between computations. One may think of it as analogous to the directory structure in an OS file system.

The designs of these structures are highly interdependent. Many aspects of the residence structure are dictated by the requirements of the startup structure, and in particular by the portability requirement on startup: computations shall be able to start immediately, regardless of 'where' an individual happens to be located at a particular time. Other aspects are determined in order to avoid using redundant copies of the same data, which means for example that information that is used jointly by several individuals shall be stored in one place and accessed by all of them, which means it is best located on a high level in the residence so it is not local to any of the individuals.

3 The SIA residence structure

3.1 The structure of the 'pioneer' individual

Since SIA residences are labelled trees, there will be several occasions where we need to illustrate trees in this text. We will 'draw' them in textual form, as in the following drawing of a minimal SIA residence:

```
GSIR
|
|- residmap.buc
|
|--- pioneer
     |
     |--- Ember
     |    |
     |    |- indivmap.buc
     |    |- invoke.buc
     |
     |--- Materiel
     |    |
     |    |- bkloader.lsp
     |    |- persist.lsp
     |    |- repro.lsp
     |    |- ad-acl.lsp
     |    |- ad-xlisp.lsp
     |    |- launch.lsp
     |    |- method.lsp
     |
     |--- Process
     |    |
     |    |--- progenit
```

```
|     |     |
|     |     |- progenit.bat
|     |     |- progenix.bat
|     |     |- progenit.sh
|     |     |
|     |     |--- Bucket
|     |     |     |
|     |     |     |- envload.buc
|     |     |     |- progenit.ifb.buc
|     |     |     |- progenit.ifm.buc
|     |     |     |- progenit.if.buc
|     |     |     |- progenit.buc
|     |     |     |- progenit.ifx.buc
|
|--- Offering
|     |-
```

The root node of this residence has the label `GSIR`. It has one single daughter node, whose label is `pioneer`, and which represents the minimal SIA individual. It has the basic capabilities for self-modification and reproduction, and is the platform on which more extensive individuals are built. When other individuals are spawned they are also placed as immediate daughter nodes of the root node of the residence.

The node labeled 'pioneer' has four daughters, labelled Ember, Materiel, Process, and Offering. All individuals have daughter nodes with these names. The use of capital first letters for their labels signifies that they represent distinct and specific functions, whereas the small first letter for individual names, such as 'pioneer', indicates that they are given names.

The individual's daughter labelled `Ember` is associated with two scripts, one of which has the label `indivmap.buc`, and so on.

Node labels in a SIA residence are of two kinds. There is a small vocabulary of *role labels* that are written with a capital first letter: `Ember`, `Materiel`, `Process`, etc. There is also an open-ended vocabulary of *object labels* that are written with small letters throughout, and that are chosen by the user, just like identifiers in a programming language are. A small number of object labels are reserved for use by the kernel, namely those that occur in the tree above.

Script labels are of the form `a.x` or `a.f.x` where `a` is an object label, and `x` is either of the following:

a) extension used by the abstract architecture: `.buc`

b) extensions and aspects that are required for the functions of the concrete implementation. This includes `.lsp` for the program files in a Lisp implementation, `.bat` and `.sh` for invocation files in Windows and Linux environments respectively, and also the aspects `.ifb`, `.ifm`, `.if`, and `.ifx` as shown above, plus in fact a few other aspects.

An implementation in another programming language will need to add scripts containing translations of the `.lsp` scripts.

(The current Windows-based implementation represents extensions and aspect-extension pairs as follows, for historical and technical reasons,

```
.buc      represented as    .lsp
.if.buc   represented as    .if
.ifb.buc  represented as    .ifb
.ifm.buc  represented as    .ifm
.ifx.buc  represented as    .ifx
```

We intend to remove the first transformation so that `.buc` scripts are represented by the `.buc` extension in the implementation, but the other representations will probably remain).

3.2 Additional Individuals: the LSIR Residence

In order to support a broad variety of applications, the SIA design uses a structured set of individuals where some are extensions of others. At present (autumn, 2003) the main version of SIA is a residence called LSIR containing five individuals, called `pioneer`, `bas`, `bas-reso`, `bas-ticc`, and ¿¿¿¿wizard-1 , respectively. Bas is a direct descendant of pioneer; wizard-2, bas-reso, and bas-ticc are direct descendants of bas.

The purpose and use of these are as follows. Pioneer is the kernel individual which will be preserved as it is, so that it can be used as the starting-point for other lines of development if needed. The individual bas contains a number of additions over pioneer that are considered to be useful in a large number of applications, such as the introduction of a type system for objects in buckets. The individual wizard-1 is a repository for 'software modules', called offerings. An offering is a package defining installation procedures that individuals can invoke in order to extend themselves. The individual bas and its descendants also contains facilities for defining ("exporting") and for installing ("importing") offerings.

The bas-reso individual contains definitions for information resources, such as documents and websites. For example, it contains the software that was used for editing and formatting the present article.

The bas-ticc individual contains facilities for simulations that are used to create simulated environments for agents.

The present article will only describe the contents of `pioneer`, that is, the principles and the details of the self-modification and reproduction machinery, as well as the design for absolute portability between computing environments. The other individuals will be described in additional reports.

4 Computations in SIA

4.1 The Working States

According to our generic model for self-modifying machines, a *computation* in SIA uses a system state and a working state. The system state or residence always follows the general pattern of the residence above, but it typically contains additional individuals besides `pioneer`, additional processes besides `progenit`, additional materiel scripts, and so on. We shall now define the working state.

The working state uses the following types of atomic expressions:

- *Role symbols, object symbols*, and *extensions*, as used in the residence. Role symbols will be written with initial capital letter and the remaining ones small letters. Object symbols will be written with all small letters (usually) or all capital letters (only for symbols representing residences). Extensions will be written with small letters only, and in the running text they will often be marked by an initial point as a reminder, for example `.buc` or `.lsp`. Digits and the characters _, -, plus, and equal are also allowed in object symbols. These kinds of expressions are jointly called *symbols*.
- *System parameters*, which are written with small letters but with an asterisk (*) as the first and the last character, for example `*process*`.
- *Strings*, which are written surrounded by `‘‘` signs, for example `‘‘This is a string’’`.

A *SIA expression* is an atomic expression or a composite expression obtained recursively by forming lists surrounded by parentheses, for example `(this is a list)`.

A *working state* is a mapping from system parameters and pairs of symbols, to SIA expressions. If w is a working state, then we write $w(p)$ for the value assigned by w to the system parameter p, and $w(i, a)$ for the value assigned to the pair of the two symbols i and a.

Working states are partial mappings, that is, they do not assign values to all system parameters and all pairs of symbols. They are extended to complete mappings by defining the value as the empty list `()` for those argument(s) not otherwise assigned a value. This object is also written `NIL`. It is obviously an atomic expression, and as such it is a fourth type that is disjoint from the three types mentioned above. `NIL` can not be an argument of w.

4.2 SIA commands

SIA commands are written as lists, for example

```
(set (capital sweden) stockholm)
```

The interpreter `I` defines how the pair of a system state s and a working state w is updated by a SIA command c. The arguments `capital`, `sweden`, and `stockholm` in the example are interpreted as constants, i.e. they evaluate to themselves. The opposite case, where an argument is to be evaluated before use, is represented as `(? f)` where `f` is a SIA form. The syntax and semantics of SIA forms is partly defined below. For the full definition please refer to the SIA system documentation.

Commands that access or update the residence s need to refer to particular nodes in it. This is done using *paths*. A path h is a sequence whose elements are either of role symbols, object symbols, or the special symbol `UP`. Such a path identifies an arbitrary node relative to the *base node* that is used in the computation. By convention, every computation uses a base node that is located as `(i Process p)` relative to the root of the residence, where `i` and `p` are object symbols. The node located at `(pioneer Process progenit)` is the only possible base node in the kernel residence showed above, for example.

The choice of base node is specified during a computation using the two parameters `*individual*` and `*process*`. Every computation has a *startup phase* where the values of a number of parameters and property pairs are set, and these parameters belong to those set during startup.

The function $target(n, h)$ determines a target node n' if n is a given node and h is a path, and is defined in the natural way as follows:

```
target(n, NIL) = n
target(n, (UP h2 h3 ...)) = target(n', (h2 h3 ...))
    where n' is the mother node of n
target(n, (h1 h2 h3 ...)) = target(n', (h2 h3 ...))
    where n' is the descendant of n whose label
    is h1, provided that h1 is different from UP
target(NIL, h) = NIL
```

The following are the basic commands and how they update a computational state `(s, w)` when `b` is the base node of the computation at hand:

```
(set p v)
    where p is a parameter: modifies w so that
    w(p) = v, otherwise s and w unchanged

(set (i a) v)
    where a and i are symbols: modifies w so that
    w(i,a) = v, otherwise s and w unchanged

(mknode h)
    where h is a path: adds whatever nodes to s
    that may be necessary so that target(b,h)
    exists (is different from NIL)

(loads h a x)
    where h is a path, a is an object symbol,
    and x is an extension: executes the sequence
    of commands stored in the script with the
    label a.x and attached to the node at
    target(b,h)

(loadm a)
    defined as (loads (UP UP Materiel) a lsp)
    This command is used for loading program
    modules in the host language where SIA has
    been implemented, for example in Lisp in our
    case

(curbk a)
    sets a (a symbol) to be the current bucket

(loadbk)
    defined as (loads (Bucket) a buc) where a is
    the current bucket.
    This command is normally used for invoking
    scripts that perform set operations, loadm
    operations, or recursively perform new loadbk
    operations. The total effect of the loadbk
```

```
    operation shall therefore be to update the
    working state w using information in the
    system state s.

(writebk)
    This command is symmetric to loadbk: it defines
    or changes the contents of the script that the
    current state assigns to the node at
       target(b, (Bucket))
    and with the label a.buc, where a is the
    current bucket set by curbk. The new contents
    are such that whenever the (loadbk) operation
    is later on performed with the same base node
    and current bucket, then assignments are made
    in the working state w that reestablish
    assignments that were in place when the
    (writebk) operation was performed.
```

These are all the commands that are essentially needed for the basic processes in the SIA architecture, that is, as the basis for self-modification and for reproduction of individuals and of processes. However, besides the commands we must also specify the mechanism for the `writebk` command, which relies on the use of *prescriptions*.

4.3 SIA Prescriptions

A *prescription* is a SIA expression that specifies how to form the contents of a script, to be used in particular by the operation `writebk`. An example will show how this works. Suppose the user performs the following operations in sequence:

```
(curbk nordic)
(set (bucket nordic) ((prop nordic bucket)
                      (prop sweden capital)
                      (prop denmark capital)))
(set (capital sweden) stockholm)
(set (capital denmark) copenhagen)
(writebk)
```

The effect of this will be to create a script whose label is `nordic.buc` and containing the middle three of these five SIA commands. In general, the operation `(writebk)` is determined by the prescription $w(bucket, a)$ where a is the current bucket. The prescription shall be a list of *prescription items* which are processed as follows.

```
   (prop a i1 i2 ... ik)
       causes expressions of the form
           (set (ij a) w(ij, a))
       to be added at the end of the script being
       produced, for each ij between i1 and ik

   (const p)
       causes an expression of the form
           (set p w(p))
       to be added at the end of the script being
```

```
    produced

(const p v)
    causes an expression of the form
        (set p v)
    to be added at the end of the script being
    produced

(loadm a)
    causes an expression (loadm a) to be added
    at the end of the script being produced

(loadb a)
    similar to (loadm a) - writes an expression
    that causes the bucket a to be loaded

(loads h a x)
    similar to (loadm a)
```

Notice how the prescription associated with `(bucket nordic)` requests the generated script to contain an assignment exactly to `(bucket nordic)`. In this way the prescription will cause itself to be added to the generated script, so that it will be reconstituted when that script is loaded.

This design using scripts and prescriptions has been modelled on the "make-file" feature in Interlisp. (This feature is not to be confused with the feature with the same name in other computational systems).

In general, the idea is to make it possible for software in a SIA architecture to define parts of the residence structure within its working state, either from scratch or being loading and modifying those structures, and then to 'write' the desired system parts to the residence again. System updates are therefore supported in a structured way.

4.4 Variable components in prescriptions and in scripts

A prescription is a sequence of second-order commands: when it is executed, it results in a new sequence of commands (the script) which in turn will be executed when the script is 'loaded' e.g. using the `loadb` command. The simplest form of prescription entries, and the simplest form of script commands represent their arguments explicitly. However there are also situations where one wishes to evaluate one or more of the arguments, either at the time when the prescription is used for generating a script, or at the time when the script is loaded into a computation. These two cases are handled using the special symbols $ and `?`, respectively.

The loading-time evaluation operator `?` is used as follows. If a script should contain the following commands

```
(set *process* myprocess)
(set *newprocess* (? *process*))
```

then the value assigned to the system parameter `*newprocess*` is obtained by looking up the value of the system parameter `*process*`. The following alternatives are available when the `?` operator is used for the third argument of the set command:

```
(? p)       where p is a system parameter -
            evaluated as w(p)

(? (i a))   where i and a are object symbols -
            evaluated as w(i,a)

(? ip x1 x2 ... xn)   where ip is an object
            symbol representing an *interpreter* -
            the value is obtained by checking the
            arguments (x1 x2 ... xn), but not
            their recursive sub-expressions for
            occurrences of ?, and then giving
            the list of the evaluated arguments
            to the interpreter named by ip. This
            handle is intended to be used for
            example for evaluating using a more
            or less specialized theorem-prover.
```

A special case of the third form is:

```
 (? lisp <form>)   where <form> is a
            CommonLisp expression - the form is
            evaluated by the Lisp interpreter.
```

The scripting-time operator \$ is similar except of course that it is used in prescriptions and not in scripts. It can occur in the following ways in prescription entries:

```
(prop a i1 i2 ... ik)
    Each of the element a, i1, i2, ... ik can
    be an expression of the form ($ ...) which
    should evaluate to an object symbol.

(const p v)
    The element p can be a $-expression which
    evaluates to an object symbol. The element v
    can likewise be a $-expression which
    evaluates to an arbitrary SIA expression.

(const p)
    This case can now be understood as an
    abbreviation for (const p ($ p))

The single arguments of loadm and loadb commands
can likewise be $-expressions.

(loads (h1 h2 ... hn) a x)
    Each one of the expressions h1, h2, ... hn, a,
    and x can be a $-expression with an appropriate
    value.
```

If ?-expressions occur in prescription elements, then they are placed as is in the script being produced. However, if the immediate argument of a ?-expression is a \$-expression, then the latter is replaced by its value as an argument of ?. Having a \$-expression with a ?-expression as an explicit argument is not possible (except possibly in special cases when the

$-expression invokes an interpreter). On the other hand, it is possible for a $-expression to evaluate into a ?-expression which is then placed in the script being generated.

5 The Startup Structure

We shift now to the structure of the startup procedure when a computation is started in a SIA individual. The proper design of this procedure is important for ensuring the mobility of individuals, but it has influenced the residence structure, and it is also significant for the reproduction mechanisms. Reproduction of individuals and processes is of course a stronger kind of startup. Also, some parts of reproduction functionality occur during the first startup of a new individual and process, much like some installation 'wizards' in ordinary software arrange that some of the installation work happens the first time the newly installed software is used.

5.1 The Startup Invocation Structure

The essential invocation structure is as follows. We use the same diagram as for the residence structure, but now the tree represents dynamic invocations and not position in the quasistatic tree. We specify it for one particular entry point, namely a Windows-style `.bat` file that calls the ACL system with the command

```
alisp Bucket/admin.ifm.buc
```

in order to start a computation with the runmode called `admin`. Notice that the base node for the computation is a process node, for example `(pioneer Process progenit)`. The invocations go as follows:

```
admin.bat
|
|--- admin.ifm.buc
      |
      |--- bkloader.lsp
      |
      |--- admin.if.buc
      |     |
      |     |--- invoke.buc
      |     |
      |     |--- envload.buc
      |     |     |
      |     |     |- residmap.buc
      |     |     |- indivmap.buc
      |     |     |- method.lsp
      |     |
      |     |--- (use startup 'methods')
      |     |     |
      |     |     |- ad-acl.lsp   (adapt to
      |     |                variety of Lisp)
      |     |     |- localmap.lsp
      |     |
      |     |--- admin.buc       (runmode)
```

```
|     |     |
|     |     |--- progenit.buc
|     |     |     |
|     |     |     |- persist.lsp
|     |     |     |- repro.lsp
|
|--- launch.lsp
```

The file or script `admin.ifm.buc` is called the *first loader script*. Such a script must do two things initially:

- Load some initial definitions whereby subsequent, programming language independent SIA script files can be loaded
- Set some systems parameters that specify the present computational environment

and then let the startup proceed using other scripts that are independent of the environment and that can be shared between several entry points. Some of the systems parameters set by the first loader script are intrinsic to it, and different first loader scripts set them differently. This applies for example to the parameter that specifies which OS is being used. Other ones of those system parameters have to be determined e.g. by asking the operating system.

A complete software individual contains several OS script files and their respective first loader scripts, which is how the individual achieves immediate portability so it is always prepared for running in different environments. These different entry points converge quickly to invoking the script `admin.if.buc` for the chosen runmode `admin`, as well as the scripts that are in turn invoked by it. These scripts are in SIA script notation, so they can be used by all entry points. The information is spread on several scripts according to where it is to be shared: some is common to the entire residence (`residmap.buc`), some is common to the individual (`indivmap.buc`), some pertains to the atelier (more about ateliers below)(`localmap.buc`), some is specific to the variety of Lisp interpreter being used (`ad-acl.lsp` for the ACL variety), and so on.

Besides the initial OS script file, for example a `.bat` file, these scripts are of two main types: `.buc` scripts and `.lsp` scripts. The latter contain ordinary Lisp function definitions, and are assumed to be manually written and maintained. The `.buc` scripts, on the other hand, and generated and regenerated from the SIA computations as directed by the SIA prescriptions, and manual interventions into them are nontrivial since they may violate consistency within the file.

Notice, however, that it is the `.buc` "bucket" files that tend to be the superior ones in the startup invocation hierarchy, and that it is they that invoke the `.lsp` files, and not vice versa. This is the key to the system's self-modification capability: by changing the data in the bucket files it is possible to change what happens during startup. In order that it shall be practically possible and manageable for the system to thus modify itself, it is essential that the expressiveness of the data in the bucket files is on the right level: not too rich, not too poor. One important result of the present experimental implementation will be to find out whether we have found the right balance in that respect.

Bucket scripts are usually sequences of SIA commands, but with the exception of first loader scripts which use the host programming language. The significant difference between bucket scripts (`.buc`) and materiel scripts (`.lsp`) is that the latter are updated and maintained by the software in the individual.

The bucket script `admin.ifm.buc` is therefore a sequence of Lisp forms, and has the following contents:

```
(load "../../Materiel/bkloader.lsp")
(setq *lispvariant* 'alisp)
(setq *os-family* 'windows)
(setq *debugmode* 'nil)
(loadbuckfile "../progenit/Bucket/admin.if")
(load "../../Materiel/launch.lsp")
```

This script loads the `bkloader` script for definitions of how to load SIA scripts; then it sets a few system parameters, then it loads the `.if.buc` script that defines the startup, and finally it loads the materiel file `launch.lsp` which is runmode-specific and defines how to conduct the communication with the user or with external clients.

The `admin.ifm.buc` first-loader script is written in Lisp because it is loaded into a naked Lisp system that has not yet loaded the code for loading SIA scripts - that is what the file `bkloader.lsp` does. Anyway, the remainder of this script after the first item would have the following expression as SIA commands:

```
(set *lispvariant* alisp)
(set *os-family* windows)
(set *debugmode* nil)
(loads (UP progenit Bucket) admin if)
(loadm launch)
```

Runmode-defining buckets have particular properties which specify which materiel files to use in the place of `bkloader` and `launch`, as were used here. Selecting other values for these properties in the runmode's bucket will lead to other materiel files being used in the place of the ones quoted here. In several cases, extensions to the most elementary system have been defined by introducting a replacement for the launch file.

A comment about a technical detail, which however illustrates the complexities that lay behind the design choices in the residence structure and the startup structure. It may seem impractical to have the invocation of `launch.lsp` or its replacement as a separate entry in all first loader files, such as `admin.ifm`. Why not have it instead in the shared script `admin.if`, which would avoid repeating the invocation in all first loader files? The reason is that in some system variants the file `launch.lsp` is kept open (in the file system sense) during the entire computation, which makes it impossible to rewrite it during that computation. If the invocation of launch were part of `admin.if` then the latter could also not be rewritten during a computation, which would be a problem since it contains parameters that are sometimes changed during computations, so that the file needs to be rewritten on occasions. The script `admin.ifm` may also need to be rewritten some times, but much more rarely since it has so few contents.

How does one replace the file `launch.lsp` by a substitute in order to obtain alternative or extended system behavior? This is not done by changing the contents of the file - doing so would be bad practice. Instead, a new file with a new name is introduced, for example `inlaunch.lsp`. One of the properties that is defined in the runmode script `admin.buc` is what shall be the 'executive'. In the original system kernel this property is set to `launch` but if one changes it, then later regenerations of the script `admin.ifm.buc` for the runmode at hand, and for its descendants, will contain `(loadm inlaunch)` instead of `(loadm launch)` in its startup procedure.

The appendix shows the contents of other major scripts that are involved in the startup sequence. The reader may not wish to study every detail in them, but we show them in order to emphasize that the actual kernel structure is quite small. This is the result of a long period of refinement in order to obtain maximal functionality from a structure of minimal size.

5.2 The use of SIA methods

One of the facilities for flexibility and robustness is the use of *SIA methods* already in the low levels of the system, such as in in the startup procedure, where methods are used for identifying aspects of the current computational environment such as, for example, the identifier of the host on which the computation is performed. The machinery for defining and using methods, as well as methods for these particular purposes, are defined in the materiel file `method.lsp`, which is invoked from the script called `envload.buc`. Once introduced, methods are used for several other purposes in the kernel, and they are available for use in applications.

5.3 Application specific individuals and processes

One SIA individual may contain several *processes* which are represented by neighboring subtrees under the `Process` node in the individual. The idea is that computations take place within processes, and each process can have one (but at most one) computation running at any one time. Each process may in turn contain a number of *buckets* under its `Bucket` node. In general, buckets are used for storing data that is local to the process, between the computations of the process. That is, a new computation may fetch information from the buckets of its own process, operate on them, deposit resulting information in those buckets, and terminate the computation. The results are then available to later computations.

We discussed by way of introduction the longevity requirement on true artificial intelligence systems. Philosophically we take the perspective that a process in the Software Individuals Architecture *is* really the subtree, or directory structure, that represents that process within the individual which in turn is within the residence. This is the structure that persists for a long time, while it is also being to modify 'itself' during that time. Computations are periods of activity for the individual, but they *are* not the individual nor a part of it.

Some of the buckets are used for *runmodes*. If a process has several runmodes, it is because there may be different ways of operating the process, for example in user command-line mode, or with a graphical interface, or via a web server-browser connection. Each runmode `r` has a bucket script `r.buc`, like other buckets, but in addition it has some other scripts such as `r.if.buc`, `r.bat`, etc which are needed only for those buckets that define runmodes.

Each process starts with a default runmode which has the same name as the process itself. Additional runmodes obtain other names.

Every new-bred individual contains one single process, called `progenit`, with one single runmode which consequently is also called `progenit`. This process may be used to *spawn* additional processes for that individual. Those additional processes may then also create additional runmodes for themselves. In general it is not recommended to add more runmodes to the progenit process, because of its role in the reproduction process.

When applications are built on the SIA platform, it is intended that they shall be realized as additional individuals and not within the pioneer individual, and furthermore as additional processes within those individuals and not in their progenit process. The `pioneer` individual and the `progenit` process serve to preserve original genetic material.

Processes other than `progenit` follow the same pattern, with one important difference: a runmode script e.g. `admin.buc` has the following contents in a minimal process:

```
(SET (BUCKET ADMIN)
     ((PROP ADMIN BUCKET BUCKET-WORD STARTER
            EXECUTIVE SPAWNER STARTUP INIT-PROCESS)
      (LOADB PROGENIT)))
(SET (BUCKET-WORD ADMIN) "admin")
(SET (STARTER ADMIN) "bkloader")
(SET (EXECUTIVE ADMIN) "launch")
(SET (SPAWNER ADMIN) PROGENIT)
(SET (STARTUP ADMIN)
     ((LOAD-LISPVAR-SPECIFIC) (IDENTIFY-RESIDENCE)
      (IDENTIFY-HOST) (IDENTIFY-ATELIER)
      (IDENTIFY-CLONE) (REPORT-COMPUTATION-CONTEXT)))
(SET (INIT-PROCESS ADMIN) NIL)
(LOADB PROGENIT)
```

Compared to the contents of `progenit.buc` that are shown in the appendix, the significant difference here is that the last form causes the neighbor bucket `progenit.buc` to be loaded, and it in turn loads the materiel scripts `persist` and `repro` that implement key functionality. In addition, a few additional properties are assigned to the process name, such as a `SPAWNER` property, specifying who spawned the present process. Once this framework has been set up properly, the buckets for the new application oriented processesd such as `admin.buc`, can include additional loading commands for loading materiel and bucket scripts that provide additional programs and data.

If there are several levels of spawning, so that 'progenit' spawns 'admin' which spawns 'user', and so on, then there will be a corresponding recursion of bucket loading operations during startup. The effect will be that the bucket file of the original ancester is loaded chronologically first, and with it the materiel files that it invokes, and then it continues in the same way up to more recent ancestors.

6 External Facilities

Although one basic idea in the SIA architecture is to make each residence as self-contained as possible, it is not possible to carry out that principle completely. Sometimes SIA individuals need to refer to facilities that can not be placed within the residence, for example, database systems. Those other facilities may even be located at other hosts so they are accessed over the net. Furthermore, if a SIA individual is located on a remote server, then it may not be practical to put all auxiliary file data there during a computation; one may wish to use local files on the host at hand.

Finally, many SIA applications require the use of a large and complex programming system, such as the Allegro Common Lisp (ACL) in the case of Lisp implementations. It is not possible to have a copy of the entire ACL system within a SIA residence, so the ACL system must be considered as an external facility as well.

The conceptual device used in SIA for coordinating the use of external facilities is called *ateliers*.

6.1 Ateliers

The basic idea is as follows. Suppose a residence containing one or more SIA individuals is kept on a detachable memory device, for example a USB-connecting solid-state (”pendrive”) memory. When that memory device is attached to a host in order to perform runs of the individual(s) in the residence, then the host must also contain an *atelier* that the individual can use. Conceptually, it is as though the software individual is ’visiting’ the host and using an atelier in that host for doing its work which is the computation.

The atelier is a directory structure that provides two things: information about the availability and location of facilities in the host that the individual can use, and space for temporary data that the individual may need. In particular, the atelier contains information about the location of the main Lisp system (for example Allegro Common Lisp) in the host.

Ateliers are an open-ended resource, so different applications built on the SIA platform can make different requirements on the ateliers that they ’visit’.

6.2 The startup procedure using an auxiliary Lisp interpreter

Computation for a SIA individual requires an interpreter, for example a Lisp interpreter. Ideally this interpreter should be included in the residence, so that if the residence is moved to another host one still has the interpreter there, and in a location that can be referenced in a standard way from the base node where computations start. In practice, however, it is not possible to include an entire, large Lisp system such as the Allegro Common Lisp within each residence.

In order to proceed systematically, it is therefore necessary to consider the ACL system as an external facility, and the ateliers will specify the location of the ACL system in the host at hand. How then are the computations started?

One possibility is to use path definitions in the operating system, so that each host environment knows where "its" Lisp system is located. We dislike this solution because then some of the essential information is not under easy control by the software individual itself, which violates the principle of self-sufficiency in individuals.

The Windows variant of the present implementation offers another solution, namely the use of the Xlisp interpreter as an auxilary system. The Xlisp interpreter is small (311 KB) and can easily be included in every residence. The startup procedure for a run with a SIA individual contains the following steps, therefore:

- the process starts running in a mini-interpreter (Xlisp) which itself is included in the residence
- the process run identifies which host it is on
- the process then decides which atelier it wishes to use on that host, using either information that it carries with it (it may have a preferred atelier on each of several hosts) or a default provided by the host
- the process loads information contained in the selected atelier, including information about where the main Lisp is located
- the process starts a run using the main Lisp system and with information that is specific to the process. This run also loads the atelier information
- later on during the run, it is able to use the atelier for additional local information and for temporary memory.

However, for some purposes it is sufficient to use the smaller Lisp system, in which case the step of invoking the main Lisp is not necessary and the entire job can be done in the Xlisp.

7 Reproduction

The term *reproduction* is used here for three related operations in SIA: the *breeding* of new individuals, the *spawning* of new processes within an individual, and the *definition* of new runmodes within a process.

The design choices that have been explained for the residence structure and the invocation structure are essential for making it easy to implement reproduction. It is particularly significant that we have data scripts (the bucket scripts) that invoke conventional program scripts (the Lisp function definition files), because then the basic strategy for reproduction is as follows:

- Generate the bucket scripts for the new individual
- Create a new subtree in the residence (a new subdirectory structure, in terms of implementation) and copy the bucket scripts into it
- Copy program files (`.lsp` files) to the nodes of the new individual

The first one of these steps is supported by the operations that have been introduced above, and the other two are straightforward.

7.1 Elementary reproduction

Elementary reproduction uses the following commands. The basic sequence for defining a new bucket is in general:

```
(curbk newbucket)
(initbk)
  ... add properties to the bucket using
      set commands
(writebk)
```

If the bucket is to be a runmode, then the sequence is:

```
(curbk newmode)
(initbk)
(modebk)
  ... add properties to the bucket
(writebk)
```

If the bucket is to be a process, then the sequence is:

```
(curbk newproc)
(initbk)
(procbk)
  ... add properties to the bucket
(spawn)
```

If the bucket is to be a new individual, then the sequence is:

```
(curbk newindiv)
(initbk)
(indbk)
  ... add properties to the bucket
(breed)
```

The operations `modebk`, `procbk`, and `indbk` assign default properties to the bucket symbol and add items to its prescription. These defaults can then be modified by the user before the reproduction operation proper.

The operations `spawn` and `breed` implement the work of creating new nodes in the residence structure (aka new directories) and of copying buckets and materiel in them, as specified in the introduction to this section. All essential information that is needed for this, for example the list of materiel files that are to be copied from the parent to the newbred individual is included as bucket-style data in the parent, and is also forwarded to the newbred so that it is aware of its own initial configuration.

In the case of spawning and breeding it may also be appropriate to do

```
(writebk)
```

before or after the spawn or breed operation. This has the effect of saving the description of the newly-reproduced object as a bucket in the reproducing process, for future reference.

7.2 Higher level reproduction

Higher level reproduction mechanisms are obtained by commands that generate sets of scripts, or copy sets of program files as one single operation. The actual implementation contains generic individuals that have been obtained from the minimal kernel individual called `pioneer`, incrementing it with additional materiel scripts and additional properties for individual and process symbols. These facilities provide individuals that have them with improved and higher-level capability for reproduction, and for newbred individuals to acquire information on their own.

One important facility for the exchange of software between individuals, both in general and as a way for new individuals to inform themselves, is provided by the *package* and *offering* constructs. They work as follows: a new facility is first developed within an individual, in particular by definition of new materiel files and modification of the bucket scripts so that those new materiel files are also loaded. When the extension is stable, it is used for defining a package with a SIA script which effectively tells what is to be done in order to load the package.

Once this has been done, an *offering* is defined in terms of a package, plus an installation script that specifies what needs to be done in order to make the package available to an individual that does not already have it. The individual producing the offering puts it into its `Offering` subtree, and other individuals are able to pick it up from there.

It is interesting that these steps - packaging, export of packages and offerings, and installation of offerings - have been made in a very concise manner. The basic facilities of the SIA architecture, such as the hierarchy of residences, individuals, and processes, as well as the bucket concept and the bucket vs materiel distinction, combine into a platform from which one only needs a thin additional layer in order to implement new functionality, such as relatively high-level reproduction mechanisms.

8 Related work, references

We do not know of any other current work that follows the approach described here. The closest similarity is to some of the work on mobile agents, in particular at the IBM Research Lab in Japan a few years ago. Mobile agents need to be able to 'encapsulate' or 'linearize' themselves in order to be transmitted to a new host where they can start to operate again, which is vaguely similar to how buckets are used in our system. Mobile agents also need a host environment, analogous to the ateliers in SIA. However there are many differences, such as the emphasis on mutual security in mobile agents: the agent shall not be able to hurt its host, nor shall the host be able to violate the integrity of the visiting agent. These are not important issues in our design, since we do not at this point foresee individuals going into hostile environments, or being hostile.

The absence of literature about self-modifying programs is a striking fact. We welcome of course any pointers to related work that we may have missed.

9 Appendix

The following are the full contents of the scripts in the SIA kernel that were referenced in the article.

9.1 First loader scripts

Startup of a SIA computation may start by invoking an OS script file, such as a `.bat` file in Windows or a `.sh` file in Unix. This file is defined to invoke a Lisp interpreter, or the interpreter of the programming language being used, with an argument for loading a particular *first loader script*. The first loader script must do two things:

- Load some initial definitions whereby subsequent, programming- language-independent SIA script files can be loaded
- Set some systems parameters that specify the present computational environment

Some of those systems parameters are intrinsic to the first loader script, and different first loader scripts set them differently. This applies for example to the parameter that specifies which OS is being used. Others of those system parameters have to be determined e.g. by asking the operating system.

A complete software individual contains several OS script files and their respective first loader scripts, which is how the individual achieves immediate portability so it is always prepared for running in different environments. However, these different entry points converge quickly to invoking a number of common SIA scripts, at which point the environment differences have been contained.

9.2 SIA-proper scripts

Once the first loader scripts have been loaded, all following scripts are in SIA script and prescription languages, which do not depend on any Lisp-specific ideosyncracies (except, if you will, the use of S-expressions as such). The following scripts have been mentioned.

The first SIA-proper script to be invoked during the startup process of a particular runmode `rm` is `rm.if.buc`, which has the following contents, assuming that the current process is `progenit`. (This is actually a bad choice but it suffices as an example). Lines starting with a ; are comment lines.

```
(set *runmode* rm)
(set (bucket-word rm) "rm")
(set (startup rm)
     ((load-lispvar-specific) (identify-residence)
      (identify-host) (identify-atelier)
      (identify-clone) (report-computation-context)))
(set (savebucket rm)
     ((makebat-xlisp) (makebat-acl) (makebuck-ifb)
      (makebuck-ifm) (makebuck-ifx) (makebuck-if)))
```

```
(set *process* progenit)
(set (process-word progenit) "progenit")

(set *runmode-word*
     (? (get *runmode* 'bucket-word)))
(set *process-word*
     (? (get *process* 'process-word)))

;; Load definitions of how to generate files in
;; startup chain
(loads (UP UP "Ember") "invoke" "buc")

;; Load description of the environment where
;; operating
(loadb envload)

;; Invoke methods, e.g. for tailoring to the
;; present variant of Lisp system
(lisp-eval (achieve-startup))

;; Load materiel files and other information that
;; is specific to the present process
(loads (UP (? *process-word*) "Bucket")
       (? *runmode-word*)
       "buc" )
```

This script contains a number of direct assignments, plus it loads three scripts, labelled `invoke`, `envload`, and (in the loads command) `progenit`.

The `envload` script loads information about the computation environment. This information is located in scripts in higher levels of the residence structure, since much of it is shared between several processes or individuals. The script has the following contents:

```
(set (bucket envload)
     ((prop envload bucket bucket-word)
      (const *envload-ok* nil)
      (const *startup-phase* t)
      (loads (UP UP UP) "residmap" "lsp")
      (loads (UP UP "indivmap") "indivmap" "lsp")
      (loadm "method")
      (const *envload-ok* t)))
(set (bucket-word envload) "envload")
(set *envload-ok* nil)
(set *startup-phase* t)
(loads (UP UP UP) "residmap" "lsp")
(loads (UP UP "indivmap") "indivmap" "lsp")
(loadm "method")
(set *envload-ok* t)
```

The last command in the script `rm.if.buc` loads the script called `rm.buc`. If the runmode is also `progenit`, which is the most elementary case, then the following script is loaded:

```
(set (bucket progenit)
     ((prop progenit bucket bucket-word process-word
```

```
       starter executive spawner startup savebucket)
      (loadm "persist") (loadm "repro")))
(set (bucket-word progenit) "progenit")
(set (process-word progenit) "progenit")
(set (starter progenit) "bkloader")
(set (executive progenit) "launch")
(set (spawner progenit) nil)
(set (startup progenit)
     ((load-lispvar-specific) (identify-residence)
      (identify-host) (identify-atelier)
      (identify-clone) (report-computation-context)))
(set (savebucket progenit)
     ((makebat-xlisp) (makebat-acl) (makebuck-ifb)
      (makebuck-ifm) (makebuck-ifx) (makebuck-if)))
(loadm "persist")
(loadm "repro")
```

The essential operations here are to load the two materiel files called `persist` and `repro`. Both of these belong to the very kernel of the system. The file `persist` contains definitions for handling buckets, plus some minor other definitions. The file `repro` contains definitions for spawning and breeding. It can be omitted for processes that do not need the capability of reproduction, but it is included in the standard kernel.

If the runmode is other than `progenit` then its corresponding script will load the `progenit.buc` bucket in the same process *instead of* loading `persist` and `repro` directly, as described in section 5.3, providing one of the inheritance mechanisms in the reproduction machinery.

9.3 Script-generating scripts

The various scripts that have been defined so far are invoked, and invoke each other during the startup phase, and they have been designed to be interpreted fairly directly. They have to be regenerated from time to time, in particular as part of the reproduction processes, and when some of the system parameters have been changed. Therefore, for each of the startup scripts there is a corresponding script-generating script, which generates the target script using a process that is essentially a kind of partial evaluation relative to the current system state and computation state. The script-generating scripts are collected in the `invoke` script in the residence hierarchy, which has the following contents in the initial system. Like everything else in the system it can be redefined in more advanced and extended individuals, and it is inherited by descendants:

```
(set (bucket invoke)
     ((prop invoke bucket)
      (prop if invokedef)
      (prop ifb invokedef)
      (prop ifm invokedef)
      (prop ifx invokedef) ))

(set (invokedef if)
       ((const *runmode* ($ *b*))
        (prop ($ *b*) bucket-word startup savebucket)
```

```
        (const *process* ($ *b*))
        (prop ($ *p*) process-word)
        (const *runmode-word*
               (? (get *runmode* 'bucket-word)))
        (const *process-word*
               (? (get *process* 'process-word)))
        (loads (UP UP "ember") "invoke" "lsp")
        (loadb envload)
        '(achieve-startup)
        (loads (UP (? *process-word*) "Bucket")
               (? *runmode-word*) "lsp" )
                ))

(set (invokedef ifb)
       (
        '(load "../../Materiel/bkloader.lsp")
        (const *runmode* ($ *b*))
        (prop ($ *b*) bucket-word)
        (const *process* ($ *p*))
        (prop ($ *p*) process-word)
        '(setq *runmode-word*
               (get *runmode* 'bucket-word))
        '(setq *process-word*
               (get *process* 'process-word))
        (const *lispvariant* xlisp)
        (const *os-family* windows)
        (const *debugmode*)
        '(loadbuck 'envload)
        '(call-siamethod '(identify-host))
        '(call-siamethod '(identify-atelier))
        '(start-acl (get 'alisp 'lisp-place)
                    *runmode-word*)
        '(report-failed-acl)
        '(load "../../Materiel/launch.lsp")
          ))

(set (invokedef ifx)
       ('(load "../../Materiel/bkloader.lsp")
        (const *lispvariant* xlisp)
        (const *os-family* windows)
        (const *debugmode*)
        (loadifbuck *b*)
        '(load "../../Materiel/launch.lsp")
            ))

(set (invokedef ifm)
       ('(load "../../Materiel/bkloader.lsp")
        (const *lispvariant* alisp)
        (const *os-family* windows)
        (const *debugmode*)
        (const *runmode* ($ *b*))
        (loadifbuck *b*)
        '(load "../../Materiel/launch.lsp")
              ))
```

9.4 Startup script using auxiliary interpreter

The use of an auxiliary Lisp interpreter for startup was described in subsection 6.2. The script for this purpose is as follows in the case of process `progenit`, runmode `rm`:

```
(load "../../Materiel/bkloader.lsp")
(setq *runmode* 'rm)
(setf (get 'rm 'bucket-word) "rm")
(setq *process* 'progenit)
(setf (get 'progenit 'process-word) '"progenit")
(setq *runmode-word* (get *runmode* 'bucket-word))
(setq *process-word* (get *process* 'process-word))
(setq *lispvariant* 'xlisp)
(setq *os-family* 'windows)
(setq *debugmode* 'nil)
(loadbuck 'envload)
(call-siamethod '(identify-host))
(call-siamethod '(identify-atelier))
(start-acl (get 'alisp 'lisp-place) *runmode-word*)
(report-failed-acl)
(load "../../Materiel/launch.lsp")
```

This script is written in Lisp and not as SIA commands because the first expression loads the definitions for loading SIA commands, and then one is already inside the Lisp interpreter. The remainder of the script would have the following expression as SIA commands:

```
(set *runmode* rm)
(set (bucket-word rm) "rm")
(set *process* progenit)
(set (process-word progenit) "progenit")
(set *runmode-word* (? (bucket-word (? *runmode*)))
(set *process-word* (? (process-word (? *runmode*)))
(set *lispvariant* xlisp)
(set *os-family* windows)
(set *debugmode* nil)
(loadb envload)
(lisp-eval (progn
    (call-siamethod '(identify-host))
    (call-siamethod '(identify-atelier))
    (start-acl (get 'alisp 'lisp-place) *runmode-word*)
    (report-failed-acl) ))
(loadm launch)
```

The command `start-acl` starts the ACL system and instructs it to load `runmode.ifm.buc` initially. The following commands are only reached if `start-acl` fails. Use of other Lisp interpreters or interpreters for other programming languages can be plugged in using an analogous construct.

The symbol-string distinction represented by the properties bucket-word and process-word are part of a scheme for managing the fact that upper case versus lower case is significant under some operating systems, such as Linux, and not fully attended to in some other operating systems. The implementation uses both a Lisp system that represents the bucket or process as such, and a string-valued property that specifies how it is represented in the directory paths.