CAISOR

Open Archival Website for
Cognitive Autonomous and Information Systems

Research Article
2005-016

Erik Sandewall

Leonardo, an Approach Towards the Consolidation of Computer Software Systems

Date of Manuscript: 2005-08-26

1 Goals and Background

The goal of the research reported here is to find an architecture for computer software that eliminates the considerable conceptual redundancy in contemporary software systems. The redundancy that we have in mind is the one where similar constructs are used in programming languages, operating systems, database systems, text formatters, www servers, webpage scripting languages such as Javascript, modelling systems such as UML, knowledge representation languages, and so on. We believe that it should be possible to design the total system in such a way that the same concepts and constructs are used for all the facilities that are today offered by separate software systems such as those mentioned, with considerable gains in cost of development, ease of learning and of use, and cross-utilization of services. We propose to use the term *consolidation* for this review and revision of contemporary software technology.

Towards this goal we have designed an experimental computing environment, called Leonardo, in order to try out and gain experience with a proposed design. This design is based on several earlier design iterations and on our experience with the use of those earlier generations of the system in a number of applications of different kinds. The present paper describes the design principles that Leonardo is based on, then describes and discusses one particular Leonardo facility that is well suited to explain its special character, namely its facility for version management, and then discusses the Leonardo design against the backdrop of the version-management example. Finally we briefly describe its implementation status.

2 Premises for the design of Leonardo

The following premises are basic to the design. Some of them are in widespread use, some are more controversial, and the systematic application of these principles in the context of a unified design results in a computation system that is quite different from the usual ones.

1. One and the same computation system shall support both the initial development, the use, and the continuous change of a software system throughout its lifetime, and the participation of its developers in that process.
2. Software is developed by groups of developers that work relatively independently of each other a large part of the time. The computation system must support the coordination of their work, providing facilities that resemble version management systems today.
3. A software system consists of *entities* that can be e.g. procedures, grammar rules in a grammar for natural language, web pages, concepts in an ontology for the application at hand, etc. Each entity has a name and a number of attributes, each of which can be a data structure or an expression in one of several languages. The software system must be able to accommodate several languages at the same time.
4. In accordance with the point of view that Donald Knuth expressed in 'Literate programming' [2], we consider that the essential structure

of software is a network of relationships between entities, rather than a hierarchy. Leonardo software attempts to represent entity descriptions and relationships between entities as explicitly as possible.

5. Entities have two representations: as text that can be read by developers, and as datastructures in the computer. These two representations must stand in a one-to-one correspondence, so that one can easily convert from one to the other.
6. A full-fledged computing environment ought to provide various kinds of user support including both automatic performance of some functions when they are needed, and tools whereby the user can perform software development operations conveniently, safely, and effectively. Smooth integration between system-driven and user-driven operations is important.
7. Both developers and computer programs therefore need to interpret, analyze, and change software, that is, the properties of entities. People use the textual representation for that purpose; programs use the datastructure representation.

Any software development environment contains, by definition, both the software in the software development tools and the software being developed. Leonardo is an *integrated* software development environment in the sense that the development tools and the software being developed are represented and treated in the same fashion, so that any part of the system can be amended by the system itself. We consider that integration in this sense is a necessary, but not a sufficient requirement for consolidation.

The total software in Leonardo is a kind of 'software individual' or self-modifying system that changes over time. Its self-modification operations are sometimes caused directly by user commands and are sometimes the results of its own, autonomous processes. Its clean and concise methods for implementing both autonomous and user-based self-modification processes is an important aspect of Leonardo.

3 Some salient aspects of the design

The Leonardo system has some similarities with incremental programming systems, such as CommonLisp, Smalltalk and Perl. It also exhibits some facilities that are simple versions of what one can otherwise find in operating systems, and some facilities that can be expected in autonomous or 'intelligent' agent systems. The combination of these facilities and a number of others (server services, version management, etc) in one concise and unified system is what gives Leonardo its unique, consolidated character. The present section describes the basic design briefly.

3.1 Entities and Entity Files

Leonardo's unified representation language is used both for representing 'programs' as usually done in programming languages, and for representing 'data' in textual form as otherwise done for example in XML. However, its approach differs significantly from both of those. An *entity file* in Leonardo is a sequence of entities, where each entity consists of the following:

- a *name*, which can be an atomic symbol or a composite expression
- a set of *attribute assignments*, each of which consists of a tag and a set-theoretic expression that is composed recursively using sets, sequences, mappings and a few other constructors, and with entity names, strings, and numbers as the elements
- a set of *property assignments*, each of which consists of a tag and a corresponding *statement* in a natural or formal language. Each statement is represented merely as a multi-line string by the kernel Leonardo system, but specific facilities or applications may assume and use a particular, tag-specific syntax for statements under that tag.

The appendix contains an example of an entity-file. The term 'file' in 'entity file' is chosen both for the traditional meaning of a file as a 'sequence', and since an entity file is usually represented as an ascii file, at least in our present implementation. This ascii file serves as the developer-oriented representation, as described above. The internal representation in the computer is as follows: each entity represents its attributes as data structures and its properties as strings; each entity-file is represented as an attribute called `contents` on its name, whose value is a sequence of the names of the entities in the file.

The 'load' and 'write' operations in Leonardo take the name of an entity-file as their single argument. The 'load' operation reads the text-file of the entity-file and constructs the corresponding data structures in memory; the 'write' operation rewrites the text file. If the contents of the entities have changed in-between then the contents of the text-file will be changed.

The appendix of this article contains an example of Leonardo entity-files in its textual representation. This appendix will be used for examples in the continued text, and its full text can also be downloaded from the Leonardo website.

3.2 Actions

The top-level control of the Leonardo system is called the *executive*. It operates in an *executive loop* which in each cycle receives *requests* from a number of sources including the user in an interactive session, from other computational processes (thereby having a built-in server capability), and from its own on-going processes. The executive adds those requests to its set of pending requests, and it initiates actions for some or all of the pending requests. The initiation of a pending request may be delayed in particular if some precondition is not satisfied. Each action is performed over a period of time and, in particular, it can execute over several of the cycles of the executive loop. The system manages its on-going actions in various ways: it keeps track of their timing, it may keep a log of actions, nontrivial requests may invoke a planning procedure for identifying a plan of actions that together can achieve the request, and so on.

The emphasis on actions is a major difference between Leonardo and conventional programming systems, in particular the interpretive ones. In Lisp, as well as the functional and other languages that have descended from it, *evaluation of expressions* is the primary operation. In principle it is supposed to be functional and 'pure', but in order to solve practical problems

well it is almost always necessary to compromise with reality and allow 'side effects' that violate the semantics of pure functional evaluation.

In Leonardo, evaluation of expressions is secondary, and the execution of actions is the primary thing. The input to the system consists of *action expressions*, which is one kind of expressions in the Leonardo representation language. The system may *invoke* an action for a particular action expression at a particular time; time being counted in terms of the system's cycle. If invoked, the action performs during at least one system cycle, and it has a distinct state for each timepoint (= cycle) including its starting timepoint and ending timepoint. Actions are able to evaluate expressions, and if a user wishes to see the value of a particular expression, she will do so by requesting a simple action whose only argument is the expression to be evaluated.

An action has a local state that changes at successive timepoints during the execution of the action. After the action has ended, its local state is preserved and can be inspected by other actions, but it is 'frozen' and can not change any more. It is used for representing the result of an action, and in particular the specific action for obtaining the value of an expression has that value as a component in its final state.

The explicit representation of actions is a characteristic feature of Leonardo. It includes the declarative representation of preconditions and effects on explicitly represented substates of the total system state, and the use of subsystems for precondition achievement and execution control even on low levels of the consolidated computation system.

3.3 High-level time management

Many of the intended uses of a Leonardo system require the system to be time-aware. The Leonardo executive therefore maintains a high-level model of time and of actions that are performed along the time axis. This model is 'high-level' in the sense that it is expressed entirely in terms of Leonardo data structures such as attributes and their values.

The core Leonardo system represents time on two levels, namely *session time* and *calendar time*. Each computational session defines a sequence of *timepoints* that are numbered from 0 and up and represent session time. They are related to physical time as follows. Physical time is assumed to be metric and can be measured e.g. in milliseconds. The physical timeline is divided into two kinds of intervals that alternate, namely timepoints and action-periods. The ending-time of a timepoint is the starting-time of the succeeding action-period, and vice versa, and each timepoint and each action-period is an interval on the physical timeline.

Consider in particular the case where the `Leonardo` system operates a read-invoke-print loop, as described above. One may then consider the physical time period where the user first thinks for a while, and then types in an action expression, as a timepoint in the `Leonardo` sense. For simple, single-cycle actions the following action-period will be the period when the action gets executed, and the next timepoint will be the physical period where the next action expression is decided and typed in.

If actions extend over several cycles, then each action-period can contain timeslots for several of the actions that go on at that time. However, the

'starting time' of an action from the point of view of the system will be the last timepoint (in our specific sense of that word) before the first action-period where the action got to execute, and similarly the 'ending time' of the action will be the first timepoint after the last action-period where the action operated.

The aforesaid applies to computational actions within the computer at hand. For robotic actions and other actions that are performed outside that computer, actions can of course usually be performed with true concurrency, and the duration of an action will be defined in terms of when and how it was controlled or observed.

Other ways of using the timeline within a session are also possible and useful. For example, in a natural-language dialog system it may be appropriate to consider each input of a spoken phrase into the system, as well as each output phrase that is produced by the system as an action in itself. In this case, `Leonardo` timepoints should characterize the starting-time and the ending-time of each input and output, instead of 'containing' such inputs. Break-ins and other situations where the user and the system perform concurrent speech acts or other communication acts can then be represented in a natural manner.

Timepoints in session time can be specified with the precision of microseconds, and are used for characterizing the beginning and end of actions within the session. Calendar time, on the other hand, is used for specifying timepoints within the life-cycle of a system, for example for version management of entity-files, or for version management of textual documents. We shall return to it in the context of version management.

4 Example: software version management in Leonardo

The Leonardo version management subsystem (LVMS) will be used as an example for explaining the character of Leonardo in a concrete manner. We assume that the user is familiar with the basic concepts of CVS[1] or other similar systems for version management. Unlike those systems, LVMS uses the entity-file structure of its own software. It has therefore been possible to implement version management as a very compact module extending the system core.

4.1 More about Leonardo entity-files

Version management in Leonardo operates on entity-files and their constituents. The appendix contains an example of an entity-file in textual form. It consists of entity descriptions, separated by lines consisting of dashes or equality signs. The following is a small example of an entity-description, for an entity called `acm`:

```
------------------------------------------------
-- acm
```

[1] `http://www.nongnu.com/cvs/`

```
[= type organization]
[= sections {sigplan sigact sigsim}]
[= fullname "Association for Computing Machinery"]

@Headline
The First Society in Computing

@Autodescription
Founded in 1947, ACM is a major force in advancing
the skills of information technology professionals
and students worldwide. Today, our 80,000 members
and the public turnto ACM for the industry's leading
Portal to Computing Literature, authoritative
publications and pioneering conferences, providing
leadership for the 21st century.

-----------------------------------------------
```

In this example, the `type`, `sections`, and `fullname` elements are attribute assignments, and the `Headline` and `Autodescription` elements are property assignments. Properties can be used for storing plain text, like in this example, but they can also be used for storing definitions e.g. in a programming language or a grammar language.

All information in a Leonardo system is represented using entity-files. Each entity-file has a name, which is by convention always the first entity in the file. In the appendix, for example, the first entity in the file is `syshistdefs` and this is also the name of that entity-file. One of its attributes is called `contents` and its value is the sequence of the names of the entities in the file, beginning with itself.

Each Leonardo 'system' is represented as a directory structure, with subdirectories on several levels[2], which is called its *home structure*. Each of its entity-files has a 'current version' that is located somewhere in this home structure, or in the home structure of a parent system from which software is inherited. Special entity-files are used as *catalogs*, mapping names of entity-files to their corresponding locations, represented as paths in the directory structure. This makes it straight-forward for a 'system' to inherit or import entity-files from other systems.

For the purpose of version management, we consider two Leonardo systems which shall be called the 'client' and the 'server'[3] The idea is that the server and the client(s) each contain their own variants of a set of entity-files, and that those in the server are the standard version. The client shall be able

[2]Many of the constructs in Leonardo are described here in terms of conventional operating system structures, such as directories and files. This is in fact overly concrete, since these constructs could as well be implemented using a database, or using entirely novel kinds of system software. We retain the conventional terms such as directory and file, however, since they are easier to grasp and they reflect the currently existing implementation.

[3]The terms 'client' and 'server' are actually somewhat inaccurate with respect to the present stage of implementation, since the update of the information in the 'server' is obtained by direct operation by the client, and not after message-passing as one would expect. This is a temporary situation, however. See also what is written below about the intrinsic server-like character of the Leonardo executive.

to compare its own entity files with those in the server, and it shall be possible to make updates both ways, based on the relative anciennity of the respective versions.

Since version management is performed on entity-files whose structure is well defined, LVMS is able to keep track of updates and time of update not only on the file level, but also on the level of entities. Version management on the level of each constituent attribute and property would be straightforward to implement, but has been postponed in order to first evaluate the experience from using the system at hand.

The operation of LVMS consists of two parts: local operations within a server or a client, and mutual update operations (sometimes called 'synchronization', inaccurately) between a client and a server. We shall describe them in sequence.

4.2 Local LVMS operations

The purpose of local LVMS operations are to allow a Leonardo system to save its software state at specific points in time, called *breakpoints*. Breakpoints are entities that represent calendar-level timepoints, which makes it possible to associate attributes with those timepoints. The names of breakpoints are formed as bp-0001, bp-0002, and so forth, and apply throughout the life-cycle of a Leonardo 'system' or individual. They have an attribute for the date and time where the breakpoint applies, and additional attributes characterizing the observed software state at that time. A basic service generates a new breakpoint on request.

We expect to add other uses of calendar timepoints, besides for software version management, and that in general they will be used for registering observations of the state of the world around the system and not merely its internal software state. This is however on the list of future work.

For each breakpoint there is a corresponding 'savestate' subdirectory containing copies of those entity-files that have been changed since the previous breakpoint.

Each entity in an entity-file contains the information about what is the most recent breakpoint where new contents for that entity are represented. This information is in the `latest-reset` attribute of the entity. Furthermore, each entity-file has the information about what is the most recent breakpoint where any of its entities has obtained a new value. This information is in the `latest-reset-entity` attribute, and is obviously the maximum of the `latest-reset` values for the entities in the file.

An entity-file is *currently preserved* iff its current version is equal to the entity-file with the same name that is stored in the directory of the breakpoint specified by the `latest-reset-entity` attribute of the entity-file in its current version. The operation of preserving the current software state therefore consists of (1) obtaining a new breakpoint; (2) identifying all entity-files in the current system that are not currently preserved, (3) changing the `latest-reset-entity` attribute of those entity-files identified in step 2, to be the new breakpoint, and similarly for the `latest-reset` attributes for all the entities in the file where a change has been detected, and (4) storing copies of those entity-files in the directory of the new breakpoint. It is clear that after this operation has been performed, all entity-files are

currently preserved, and they remain so until again some changes are made to current versions of entity-files.

Local preservation of software state is useful as a way of allowing the user to back up to an earlier version of her or his system, but it is also the natural starting-point for version update between client and server.

4.3 Client-Server LVMS operations

The basic idea for client-server LVMS operations is that each server and each client preserves its software state from time to time, as just described, and the effect of a client-server update is that the system state of the latest breakpoint in the client equals the system state of the latest breakpoint in the server. Notice that the update operation normally causes a new breakpoint to be generated in the client and/or the server, so we refer to the system state of the latest breakpoint that is present *after* the update, on the two sides.

The client-server update operation is therefore simple in principle, and goes as follows. (1) Ascertain that all entity files in both the server and the client are currently preserved. (2) Identify the breakpoints of the previous update, that is, the latest breakpoint in the client that was updated against the server, and the breakpoint in the server that it was updated against. (3) Identify those entity-files in the server, and in the client, whose most recent preservation (known by an attribute on the entity-file) occurred later than the breakpoint of the most recent update in the server or the client as the case may be. (4) If those sets of entity-files are disjoint then update in the obvious way on the file level. (5) If an entity-file has been updated in both the server and the client since the breakpoints identified in step 2, then perform the analogous operation on the level of each entity in the entity-file. (6) If there have been concurrent updates even on the entity level then allow the user to decide.

There are obvious questions about the safeness and the appropriateness of this kind of version management, in particular when it is used to 'blindly' merge software changes that are made independently by several software developers, but we will not address that issue here. The fact remains that it is used with great utility in practice.

This description of the LVMS procedure is not complete, and in particular we have omitted the description of what actions to take when entities appear or disappear between versions, and when entities are moved from one entity-file to another. Those considerations are addressed in the actual system, but they do not add anything essential to the example.

5 Discussion of the LVMS example

The LVMS example illustrates both the look-and-feel of entity-files (in their textual representation), the deeper structure of those entity-files, and as well the algorithmic processes that they lend themselves to. In particular it illustrates how Leonardo makes a full integration of what is usually separated as 'programs' and 'data'. In this section we shall discuss a number of important design issues in Leonardo against the background of this example.

5.1 Composite entities

Entities in Leonardo can be atomic or composite. The following is an example of an entity description called `(location: syshistdefs)` that occurs in a catalog entity-file, in our current system:

```
------------------------------------------------
-- (location: syshistdefs)

[= type location]
[= latest-reset nil]
[= filename "../../../leo-1/Defblock/syshistdefs"]

------------------------------------------------
```

The purpose of this entity is to specify the 'physical' location of the entity-file `syshistdefs`. It would not be appropriate to let that be an attribute of the entity `syshistdefs` itself, since then the filename would be generated into the entity-file (making it harder to move copies it around), and all other attributes of the entity `syshistdefs` would go into the catalog file.

Composite entity-names are used in several ways in the Leonardo system, even on its most basic levels as this example shows. At the same time they are a very powerful facility in a representation language; they correspond of course to 'terms' in logic and in e.g. Prolog, but they are missing in many conventional representation languages.

The use of composite entity-names is therefore one example of how the consolidation of the computing environment, as proposed in the introduction of this article, leads one to reconsider traditional concepts as to what are 'low level' and 'high level' concepts in a programming language and a programming system. Additional examples of this will be shown in the sequel.

5.2 Comparing entity-files

Some of the most basic operations of the Leonardo executive are to *load* an entity-file and to *write* one. These two operations represent the conversion between the text-file representation and the data-structure representation of a sequence of entities. They are each other's inverses, so that loading an entity-file and immediately writing it causes the file (in the operating-system sense) to be rewritten with the same contents as before. This means that if the user edits the text-file representation of an entity-file while the system is running then the file should be loaded, and if the software modifies the segment of its data-structures corresponding to an entity-file then the file should be written, in both cases to maintain consistency.

This arrangement provides an excellent basis for all software that operates on the contents of an entity-file. However, the question arises how one can best implement a program that compares two versions of the same entity-file. Certainly one can not just load them both, since then the last one to be loaded would just overwrite the other one. This need arises, in particular, when the system is going to compare the current version and the preserved version of an entity-file, or when a client is to compare its own preserved version with the one that is preserved in the server.

This need is met with a small generalization of, or 'handle' in the procedure for loading an entity-file, and a similar one for writing the file. The idea is that when an 'alternative' file version is loaded, then each entity `enti` that appears textually in the file is represented as a composite entity of the form `(f: enti)` within the system, where `f:` is a suitably chosen operator for formation of composite entities. An additional parameter for the 'load' operation allows one to specify the appropriate wrapper operator `f:`. Similarly, a parameter for the entity-file 'write' operation allows one to specify that entity-names in the file are to be unwrapped when written. These generalizations in the 'load' and 'write' operations are very minor in size, and they are quite powerful.

5.3 Look-and-feel of entity-files

We turn now to a seemingly more mundane issue, namely the syntax and look-and-feel that is used in entity-files. It follows from the above that entity-files replace both 'program files' as used in ordinary programming languages, and 'data files' as in e.g. XML, and they do not look anywhere near either of those.

Our major consideration in designing the textual format was that it should be as convenient as possible to use for people, while at the same time staying close to the data-structure representation that is used inside a reasonable implementation. We also wanted it to be able to accommodate multiple languages and formalisms (programming languages, query languages, grammar notations, etc). Finally, it should lend itself to the load/write discipline: it must be possible for the system to load an entity-file in such a way that it is able to regenerate it. This capability is fundamental as a basis for 'software apprentices' that do some of the menial tasks in cooperation with the user that does the high-level and non-trivial tasks.

On the 'data' side, our design differs from XML and XML-related formalisms in a number of ways. We insist on using normal set-theory constructs such as sets, sequences, and mappings rather than the more elaborate constructs that are used in XML. We also insist on using standard set-theory notation instead of the SGML-based markup like `<tag> ... </tag>`.

In addition to conventional set theory we distinguish between sequences and records, where for example the date of November 23, 2005, can be represented as the following record:

```
[date: 2005 11 23]
```

Leonardo expressions are formed recursively from atomic entities, strings, and numbers, by composition using composite entity-formers, sets, sequences, records, and mappings in the obvious ways. Notice that records are just one kind of expression, and the elements of records can in principle not be assigned by way of side-effects. An expression such as

```
[= age 46]
```

is an example of a *maplet* that assigns a value to e.g. an attribute. The "record type" `=` is used for forming maplets and mappings are sets of maplets.

Besides these structures, Leonardo also contains a number of conventions for representing formatted text. However, we prefer to organize separate notations for expressions, i.e. structured data, and for text, albeit with a possibility of escaping from one to the other. XML combines them into one single notation which has its historial roots in SGML, and therefore in the perceived needs of markup of text. We fail to see how its notation such as

```
<age>46</age>
```

can be motivated either from the point of readability for humans, or for processability for computers, or as systematic design.

With respect to programming-language notation, the 'BNF' tradition in the design of programming languages is extremely strong. The essence of this tradition is that entire programs shall be formed recursively, so that the same constructors can be used on all levels. The motivation for this choice is that large systems must be designed in a hierarchical fashion in order for the design and the maintenance to be manageable, which is a most reasonable point of view. However, it is not a warranted conclusion that the entire structure must be *uniformly* hierarchical, so that all levels must be formed in the same way. We can easily use different structuring methods in different layers of the hierarchy.

The major level in Leonardo, at least as present, is the Leonardo system which is a kind of 'individual'. The client and the server, in the situation that was just discussed, are separate individuals. Within the individual there are a number of entity-files; within the entity-file a number of entities. In fact it is possible to divide an entity-file into sections, each containing entities.

Each of these levels has its own specific characteristics, which are motivated by a number of practical considerations in terms of how people work with software development. Recursive structures occur on lower levels, in particular in the program expressions that are stored in the 'properties' of entities, but not on the highest levels.

The conventions for separating entities in an entity-file provide a good example. Given that an entity-file is more or less a sequence of entity-descriptions, a traditional programming language might have represented the separation as

```
end entity
begin entity POPL-06
```

and in XML style one might have

```
</entity>
<entity><name>POPL-06</name>
```

In both cases this shows the end of one entity-description and the beginning of the next one. In Leonardo the same passage is instead represented just as

```
-----------------------------------------------
-- POPL-06
```

The first representations may be more elegant, in the sense that a recursively defined syntax for an entire program can be appreciated as aesthetically pleasing. The convention used by Leonardo has advantages of other kinds.

From the point of view of the human reader that tries to orient himslf in many pages of program code, there is no question that Leonardo's way of separating the entity-descriptions is perceptually much more effective: you can easily see how the file is partitioned into entities. It also has the effect that the partitioning of a file into entity-descriptions when it is read by the file-compare program is robust with respect to syntax errors within each of those entity-descriptions.

6 Actions in Leonardo

Before leaving the LVMS example, we shall discuss one other important aspect of Leonardo that is used also for LVMS, namely, the support for cooperative action.

6.1 Autonomous actions for mundane tasks

The basic issue is as follows. The main task of the Leonardo executive is to perform actions. In doing so it maintains a log of past actions, including their starting and ending timepoints. However, the same datastructure for timepoints and past actions can also be used for storing observations of actions that are performed by other systems, or by users. This is important e.g. in robotic applications.

Actions can have a declarative representation using preconditions and post-conditions in the usual way, as well as their procedural (operational) definitions for performing or for observing the action. The executive has the capability of identifying when the preconditions of a requested action are not present. In such cases it can either refrain from executing the action, or find and perform a plan for achieving the preconditions. This type of autonomous behavior is well established in high-level autonomous agents, but not usually on the operating-system 'level' or the programming-language 'level' of contemporary software.

However, there are many examples of specific facilities on those levels that do similar things, for example, allowing the user to provide a required password for being able to fetch particular information from another host, or turning on a printer in order to obtain a requested printout. We propose that the incorporation of autonomous precondition achievement and other, similar action support can be very useful even on 'lower' levels of an integrated computing environment (to the extent that the high-level/low-level distinction is at all meaningful there).

6.2 Cooperative action between user and system

Let us return now to the version update operation between client and server. It can be seen as an action consisting of a number of sub-actions. The main action has a well-defined goal state, namely, the state where the client and the server have fully preserved current states (no unpreserved entity-files) and their respective latest breakpoints have equal contents. A number of subactions are required in order to achieve that goal state, as was described above. Some of those subactions can be performed autonomously by the

LVMS, others require user intervention, in particular for resolving concurrent updates.

Leonardo provides support in two ways for this cooperative goal-achievement or 'problem-solving' process. Firstly, the client-side latest breakpoint provides a kind of scratchpad for the work during the period when the goal has not yet been achieved. A breakpoint, viewed as a Leonardo entity, can have attributes that specify e.g. the set of locally unpreserved files, or the emerging relationship to the latest server-side breakpoint.

If a number of non-trivial updates have to be resolved in the course of using a conventional version management system, such as CVS, then the user may feel the need for a scratchpad where he can keep track of what needs to be done, and what has already been done. In LVMS it is the client-side latest breakpoint entity that serves as such a scratchpad.

Secondly, the built-in facility for *monitoring* plan execution and goal achievement can be used here. The idea is as follows. Autonomous plan execution (without user intervention) is obtained using two concurrent processes, namely, one process that generates the next action to be performed, and another process that observes the current state of the computation and (if applicable) the world, and that notices when goals and subgoals have been achieved. These observations are in turn used to guide the choice of next action. These two processes are called the actor process and the monitor process, respectively. The effect of the clean separation between them is that the individual actions that are performed by the actor do not need to report their results to the overall control of plan execution. Individual actions just do what they do, and then it is up to the monitor to observe what has happened, either as the direct results of actions by the same system, or for other reasons.

Instead of autonomous plan execution, it is sometimes appropriate to apply user-driven plan execution, where the actor process is suppressed and only the monitor process remains, together with a notification facility. This is in particular the case for version management: the monitor process for version management is able to observe the achievement of subgoals. Using it, LVMS is able to remind the user of what needs to be done, and to caution or even stop execution of actions if they are inappropriate in the current situation.

7 Software structure in Leonardo

7.1 Programming language

In the long run we would like to implement Leonardo directly on the computer hardware. However, its present implementation has been done on top of an existing language, namely CommonLisp, which in turn runs on top of an existing operating system: Linux or Windows. Similar implementations using other programming languages are of course equally possible, although we observe that the implementation is much easier in an incremental language. The structure of entity-files and the syntax for attribute values (essentially: set-theory notation, adapted to the standard 8-bit ascii character set) are independent of the choice of implementation language.

With respect to the choice of programming language for use inside the

entity-descriptions, the use of CommonLisp is merely an intermediate solution. Its major disadvantage is that it requires us to write the program code in terms of the Lisp (list structure) representation of Leonardo expressions, and not in terms of those expressions themselves, making it difficult to "quote" Leonardo expressions in-line in the program.

Development of a new, functional-style language that combines well with Leonardo data expressions would have several advantages, but the exact design of such a language remains open. Since attribute-values in Leonardo are expressed in set-theory notation, it would be natural to define a functional language where functions are defined as sets of maplets and those sets can be defined by membership condition and recursion, and not merely by enumeration.

A similar argument applies to the language(s) for defining actions. This will be a topic of another article.

7.2 Invocation records and servers

Actions are invoked using action expressions like in the following example

```
[fly-to! :agent witas-4
         :destination [geo-coord: 425 862]]
```

saying that the UAV (airborne robot) `witas-4` shall fly to the point located at geographical coordinates (425, 862). Records and action expressions use the same conventions for their 'arguments', namely that there may first be a number of untagged arguments and then a number of tag-argument pairs. The symbol `:agent` in the example is a tag. The number of untagged arguments is specific for each record type and each action verb. Record types end with a colon and action verbs with an exclamation sign.

Furthermore, this notation is actually a shorthand in the sense that in principle each action expression contains exactly one argument, which is a record. Sometimes the type of the argument record is specific to the action verbs; sometimes there can be several action verbs that share the same argument record type; some action verbs may accept several argument record types. The executive loop of the Leonardo executive reads expressions that are input from the user, consisting of action verb and arguments, constructs the appropriate argument record, and gives it to the action verb for invocation.

This design means that the characteristics of a server are built into the Leonardo executive. In each executive cycle it does not merely receive an expression that is typed in by the user, it also accepts action expressions from incoming message channels. For example, a web server in Leonardo would handle active web pages by considering the URL request as a Leonardo action invocation. The ad-hoc HTML conventions using question-marks and ampersands in URL requests would be replaced by the Leonardo record format. In this way, action invocation within a system and between systems can be handled in a uniform manner, which is one additional aspect of consolidation.

7.3 Type systems

Leonardo has a simple type system for entities and a rudimentary type system for records. Entities have one attribute called `type` whose value must be an entity of type `leotype`. Entities of type `leotype` have an attribute called `props` where the value must be a set of attribute names. Attribute names are also entities, of course. This is as much as is needed for the operations of loading and writing entity-files.

Leonardo's present type system does not provide for type hierarchies of any kind. Extensions in this direction will obviously be needed in many applications, but they are not crucial for the design of the core system so those considerations have been postponed until later.

The notion that the type system is *not* a part of the core of the architecture, and is instead an added facility constituting a higher layer of design, may seem foreign from the point of view of traditional programming-language design. It is however a natural consequence of the consolidation enterprise. Consider, for example, the arrangement that was described in the previous subsection, namely that remote invocation of web services can be done using the same invocation format as an action invocation within one Leonardo system. This contributes to the conciseness of the design and eliminates one potential, conceptual redundancy, but it means at the same time that the invocation method in the core system must be based on the assumption that type checking for the argument(s) in the invocation is done dynamically by the invokee. It would not be possible to maintain static typechecks for all possible invocation paths between Leonardo-type systems in the Internet. Instead, in the Leonardo design, the basic system uses dynamic type checking at invocation time, and static type checking of invocation paths is seen as a technique that can be used within designated applications or subsystems where the set of components and their interfaces is known statically.

We discussed above how consolidation sometimes results in a downward move of facilities from higher to lower levels of the global architecture. For typechecking there is the opposite change: we propose that it should be moved from a lower to a higher layer there.

We mentioned above the longer-term need for a programming language that connects strongly to Leonardo datastructures. In our view the kernel of that language should be type-free, for the reasons that have just been explained here, but the language should be designed in such a way that an effective type system can be overlaid on it.

7.4 Types of composite entities

An entity constructor is a function whose values are composite entities, for example the function `location` that was mentioned above. Each entity constructor is associated with a function that creates the composite entity from its arguments and automatically assigns some attributes to it, in particular the `type` attribute. The `location:` constructor always assigns the leotype `location` to the composite entity that it generates. On the other hand, the entity constructor `inparent:` that is used when two versions of the same entity-file are to be read into the same system for comparison (as described in subsection 5.2) sets the type of the constructed entity to be the same as the type of its argument. Its definition is included in the example entity-file

in the appendix and shows how the type of the composite entity is obtained as the type of the argument.

7.5 Locality of names

We have described how software in Leonardo is fragmented into a set of entity-descriptions, where each entity has a name. This suggests that all entity-names are globally defined during their use in a Leonardo system. On the other hand, one of the major reasons for having a hierarchical structure in a conventional, algorithmic programming language is in order to provide locality for names. One may ask whether this important property of such languages is lost in the Leonardo approach.

In Leonardo and in a plausible Leonardo implementation, one must distinguish between two cases: multiple uses of one single entity, and multiple entities for a single entity-name. Then by 'entity-name' we mean the textual atom that appears in the textual entity-file, for example `syshistdefs` in the appendix. By 'entity' we mean the data object that is created and used in the working system when it reads the entity-file.

There are a number of situations where one entity-name always maps to the same entity, and that entity has multiple uses corresponding to locality of names in an algorithmic language. This occurs for example for the formal variables in CommonLisp function definitions (lambda expressions), and for entities that are used as tags in the local states of durative Leonardo actions[4]. Multiple entities for a single entity-name are created using entity composition, as was discussed in section 5.2 above. This is the way to deal with, for example, the situation where one wishes to put together two subsystems that have been implemented separately and where there is a name collision for some of the entities in them.

7.6 Attaching function definitions to entities

If one starts from the point of view of classical programming, then it is natural to organize a program as a set of functions or procedures, and to let each of them be represented as an entity in the Leonardo system. However, there are also many situations where it is more natural to let some attributes in certain entity types have function definitions (lambda expressions) as values. Class definitions are an obvious example. We also use this technique for entity constructors, as shown in the definition of `inparent:` in the appendix, and for several other purposes even in the basic system.

7.7 Working with entity-files

Although Leonardo is designed to facilitate writing and using software that operates on entity-files, it remains that most of the development work in Leonardo relies on direct editing of entity-files on the text level. A common working cycle is then for the user to (1) edit the entity-file in its textual

[4]This topic is not discussed in the present article, and we refer to our other articles on Leonardo for more details.

form, (2) load that file into the Leonardo system being used, (3) write the file again, (4) check that the file so produced looks right. Besides for verification, step 3 also does some clean-up of the text so that the user does not have to attend to trivial details when the text is edited. For example, the line of dashes that separates entities is recognized by the loader regardless of the number of dashes in the line, but it is produced with standardized length by the writer for neat appearance.

A less trivial example concerns the `contents` attribute of entities that represent entity-files. (See for example the first entity in the entity-file in the appendix). This attribute contains the sequence of all entities in the file, and is essential for all operations that range over all those entities. In particular, it is used by the writer for deciding which entities to write. When entities are added to, or removed from an entity-file, it is sufficient to do that to the entity-description (between the two lines of dashes). The `contents` property is reconstructed when the file is loaded by observing what entity-descriptions actually occur in the file.

This way of operating on entity-files makes it straight-forward to implement basic service facilities. For example, the operation of sorting the entities in a file in alphabetical order is done by sorting the sequence in the `contents` attribute of the filename, followed by a write operation of the entity-file.

Entity-files for program-type information (definitions of functions, actions, etc) usually end up being fairly short, and then the described procedure is convenient. Entity-files for data, for examples 'bibtex' bibliographies often become quite large, containing a large number of entities of the same type. At some point it becomes too slow to re-read the entire file after each update of a single entity-description in it. In those cases it is more convenient to create a small file only containing the entity to be edited besides its own name, write it, text-edit it, and reload it. The definition of this action is very simple and can serve as an additional example of how it is easy to program operations on entity-files: use an editing entity of type `entity-file` e.g. called `editfile`, assign the following value to its `contents` property,

```
<editfile entity-to-be-edited>
```

write that file, text-edit it, reload it, and rewrite the main entity-file containing `entity-to-be-edited`. Writing an entity-file is instantataneous even for large files, in the present implementation.

7.8 Text preservations vs prettyprinting

There are two major differences between 'attributes' and 'properties' in the Leonardo system. First, attributes must be expressed in Leonardo's textual representation for datastructures, whereas properties can be used for expressions in the formal language of choice of the user. Secondly, attributes are converted to datastructures within the software when an entity-file is read, and the textual representation is reconstructed from the datastructure when the entity-file is written, but properties are stored as plain albeit multi-line strings by the core Leonardo system. It is then up to any program using the property to parse it according to its choice of conventions, and use the results.

For the particular case of Lisp function definitions, which are significant to the extent that CommonLisp is (presently) used as the implementation

language for Leonardo, these are represented as properties and not as attributes, which means that their indentation and other aspects of their textual appearance is retained. If a user wishes her Lisp code to be regenerated for the purpose of prettyprinting then this has to be done by a separate operation, and the same applies for other notations that are used in Leonardo properties.

7.9 Normalized and reduced entity-files

We have stated already that all software in a Leonardo system is expressed as entity-files. This raises the question of how to start the loading of the system: what about those entity-files containing the definitions of the loader and the executive loop, for example?

The answer to this is that the write operation actually produces two files (in the operating-system sense) with equal name except different extensions. Besides the *normalized* file which is illustrated in the appendices, the write operation also produces a *reduced* file with equivalent contents that can be read directly by the implementation language. In our case, the reduced file contains a sequence of Lisp S-expressions that can be evaluated by the Lisp interpreter. The startup procedure of the Leonardo system begins by loading a few reduced files, for example for the loader of normalized files, and then proceeds to load other files in normalized format. Changes to the initial files can always be done by editing their normalized text files, loading them, and rewriting them.

Changes of this kind in the most basic facilities of the system run of course a risk of leaving the system in an inconsistent intermediate state where it is not able to proceed, even if the result of all the intended changes is again consistent. Facilities for monitoring, avoiding, or repairing such problems are on the list of future work.

In large systems and in production mode usage it may also be of interest to use the reduced version of entity-files throughout. In particular, they can be given to the Lisp compiler.

7.10 Self-modification in Leonardo systems

We have already mentioned the notion that a Leonardo system is intrinsically a self-modifying system. The concept of self-modification is complex in computer science. One would expect a priori that it should be an important topic, since a characteristic feature of the von Neumann computer is that it can modify its own programs. However, a search for references to self-modification leaves disappointingly few results. Besides the special cases of genetic programming and neural networks, which although interesting in themselves are not really relevant here, traditional literature search offers very little. Google offers a number of links to webpages for programming-language courses with comments along the lines of 'here is a neat thing you can do in Lisp, but it is hard to think of a way of using it, and self-modifying programs are difficult to work with'. Furthermore, conventional programming languages are usually designed with an explicit notion that program self-modification is one of those things that the programming language should prevent its user from doing.

At the same time, self-modification is ubiquitous in actual computing systems, in particular if you consider the totality of all the software in the computer as *the program* of that computer. Compilation is of course self-modification under that definition. Installation of an additional piece of software is self-modification; installation of a patch or a new release is so as well.

We therefore have the somewhat bizarre situation that self-modification is much used but little treated from a systematic point of view. Textbooks on programming languages or software engineering typically do not mention it, or only mention it dismissively. It is therefore not surprising that software tools for software installation are called "wizards", suggesting that there is something strange and magical about them. Bringing software self-modification out into the open and treating it systematically should be on the agenda for the future, and Leonardo proposes one approach to doing exactly that.

8 Related Work

The work on Leonardo relates to several existing branches of computer science, such as agent systems, and logics for reasoning about actions, besides the principles of programming languages and systems. A full discussion of related work is not possible in the present context, and we must refer to our other publications for a more complete coverage.

The issue of "related work" for Leonardo is to a large extent a question of "alternative and similar approaches", and the best entry-points for them is often their websites, rather than specific research articles. In those cases we cite the relevant URL:s, and the list of conventional references below is quite brief as a result.

8.1 Programming languages and systems

On the programming-language arena, the most strongly 'related works' are the following. First of all, the Lisp language and system, and in particular the Interlisp system[4] and the software systems of the various Lisp machines that were designed in the 1980's. Interlisp pioneered the idea of a programming environment which has then been inherited by other languages and communities, and the Lisp machines showed that it was possible to integrate operating system and programming language in a strong way.

The Smalltalk language and system[5] has integrated concepts from Lisp and Simula[6], and seems to be the strongest follower of the Interlisp design philosophy today. The Perl language[7] shows in a modern setting how the facilities that are needed on the command level of the operating system can be extended into becoming a serious altough still special-purpose programming language.

The first use of set theory for programming was to my knowledge the SETL language[3]. `Leonardo`'s internal programming language at present is not

[5] `http://www.smalltalk.org/main/`
[6] `http://www.isima.fr/asu/`
[7] `http://www.perl.org`

covered in the present article, but it is somewhat similar to the style in the Erlang[8] language which in turn obtained it from Prolog[9].

The definition of the executive loop implements concurrency in a way where the current state of each concurrent action is open to inspection and can be referenced. By comparison, management of concurrency using 'detach' and 'resume' operators during an evaluation process requires complex stack management methods that are (intentionally) hidden from the program. The same applies for the use of continuations. The approach used by `Leonardo` may be perceived as more restrictive, but it is closer to the representation of current state in logics of actions and change, facilitating the use of planning and plan execution techniques.

8.2 Literate programming

In the approach of *literate programming*[10] there are three representations of a computer program: a source form that is edited by the programmer or developer, a "woven" representation that is adapted for reading, and a "tangled" representation in a conventional programming language that is fed to the compiler for eventual execution in the computer. This representation is chosen *because a program is best thought of as a web instead of a tree* (Knuth). The representation of software using entity-files in Leonardo is based on the same view, but takes it one step further: the entity-file is used as both the 'source' and the 'tangled' representation in literate programming. In fact, if Knuth's observation that the essential structure of a program is as a network of relationships is taken seriously, then the need for the 'tangled' representation in the target programming language can be interpreted as an artifact of that language, not warranted by the character of the software per se.

With respect to the 'woven' representation, we foresee that the entity-file representation should be complemented with a 'literate' representation that is essentially an essay about the program, within which specific entity-descriptions or parts of them have been included. It is easy to see how one could design a text formatter so that it allows formatting commands that include specific material from the entity-files.

8.3 Related work in other areas

The 'action' aspect of Leonardo is based on our earlier work in knowledge representation, on logics of action and change. This aspect has been described in another recent article, and the reader is referred to that article for more details than are given in the present article.

A number of agent systems and languages contain concepts and constructs that have been taken up in `Leonardo`, in particular RAPS[1].

[8] `http://www.erlang.org/`

[9] `http://pauillac.inria.fr/~diaz/gnu-prolog/`

[10] `http://www.literateprogramming.com`

9 Concluding discussion

9.1 The possibility

The basic idea in Leonardo is to try a new way of organizing the overall structure of the software in a computer, questioning our heritage of assumptions about the proper roles of operating systems, programming systems, and many other software artifacts. This is what we mean by 'consolidation' of the overall software structure. There are a number of reasons why consolidation is a valid enterprise at the present time.

The most fundamental reason is of course that the present structure has been around for a long time, and many things have changed meanwhile. Software strategies that were impossible for performance reasons in the personal computers of 1990, and even more unthinkable in the time-sharing systems of 1970, may now be quite acceptable. A major use of raw computing power for supporting the programmer in recent times has been to provide ever more impressive graphic capabilities. Maybe there are other ways, with greater conceptual contents and strength, for using some of those cycles.

Another fundamental reason is that software that is continuously developed and used for a long period of time tends to become overly complex. This is a well-known phenomenon on the level of a particular software project: if you have written a piece of software, developed it for some time, and then you decide to throw it away and start over, then chances are that the new version of your software will be considerably more compact and less 'bloated' than the original one – it will be 'consolidated'. This happens for a number of reasons, including that a normal way of dealing with a design deficiency is to write more code. My point is that the same principle may apply on the global scale, that is, on the total structure of software that is found in computers today, and that the current state of software technology is such that consolidation is called for.

9.2 The impossibility

If consolidation of the global software system is needed, why has it not been done already? One reason may be that the contemporary global software architecture is very robust because of the interdependencies: any new programming language must be useable together with existing operating systems, and vice versa, and similarly for database systems and other major artifacts. This robustness does not invalidate the need for consolidation, it just makes that the consolidation is delayed and becomes more dramatic when it comes.

A possible objection against consolidation and therefore against the Leonardo project, and one that I have encountered repeatedly, is that it is entirely unrealistic. How could you possibly imagine that the world will change to a new software solution for the services now provied by operating system, programming languages, database systems, and all the rest, all at the same time? Previous attempts to establish new generations of software technology with a bang have not been overly successful.

My answer to this is by reference to the Algol 60 language. During its

active life it was proposed as an alternative to Fortran (more specifically, Fortran II and Fortran IV), and it did not reach very large acceptance due to factors such as performance, and the existence of large program libraries in Fortran. However, a number of subsequent languages such as Pascal, ADA, and Java have continued many of the concepts and solutions that were first established in Algol 60. In that particular sense Algol 60 must be considered as a long-term success. Any attempt to formulate an alternative to the contemporary global software architecture, any attempt at consolidation, and any evaluation of such an attempt should adopt a similar long-term perspective.

10 Design History and Implementation

Our present `Leonardo` system has been implemented in CommonLisp, starting with the Leonardo executive, the functions for reading and printing `Leonardo` expressions and entity-descriptions, and for defining and executing actions in the framework of the executive. A number of facilities have been migrated from our earlier software base of "software individuals" to the Leonardo system, including facilities for document processing, website management, and bibliography management. For example, the present article has been prepared within the Leonardo document processing facility, and CASL group website[11], which also contains material about Leonardo, is administrated using Leonardo.

The concepts that have been synthesized into `Leonardo` have been present in our own work during a long time and have evolved gradually. Many aspects of the language design, including the proposed design of agents, have also been influenced by the experience of building a robotic dialog system and its auxiliary robot simulator, as a part of the WITAS project[12]. This applies in particular for the view of the top-level executive of the system which bears some resemblance with the robot simulator that was implemented as a tool for the development of the dialog system.

The long-term research background for the present work is documented on the CAISOR website[13].

References

[1] R. James Firby. Task networks for controlling continuous processes. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 62–69, 1994.

[2] Donald Knuth. Literate programming. *The Computer Journal*, 27:97–111, 1984.

[3] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[4] Warren Teitelman. Toward a programming laboratory. In *International Joint Conference on Artificial Intelligence*, pages 1–8, 1969.

[11] `http://www.ida.liu.se/ext/casl/`

[12] `http://www.ida.liu.se/ext/witas/`

[13] `http://www.ida.liu.se/ext/caisor/`

Acknowledgements

We acknowledge the contributions to the present work that Egil Möller and Peter Andersson have done by introducing the use of version management systems in our research group.

We also gratefully acknowledge the following support. The WITAS project has been sponsored in full by a grant from the Knut and Alice Wallenberg Foundation. The concluding step of work leading to the present article was done while the author was visiting the Contraintes research group in the INRIA Research Center at Rocquencourt (Paris), France.

```
APPENDIX: An example of an entity-file in Leonardo format
The name of the file is 'syshistdefs'

------------------------------------------------------------
-- syshistdefs

[= type entityfile]
[= latest-reset bp-0001]
[= contents <syshistdefs syshistdefs-1 compl create-bp fixf
fixfiles fixl inparent loc-compare makinparent makinparent-def
nexbp read-entity-file readent-cur readent-save
savestate-file symbarg1 writesave syshistdefs-2 after-bp
after-bp2 consider-update do-update latest-bp-in-parent
load-parent-save load-parent-syshist loadpl loadpsh princp
princt vadw version-admin-down write-parent-save syshistdefs-3
check-udx udxa cludx syshistdefs-4 goalagree inver>]
[= sections <syshistdefs-1 syshistdefs-2 syshistdefs-3
syshistdefs-4>]
[= latest-reset-entity bp-0002]

============================================================
-- syshistdefs-1

[= type section]
[= latest-reset nil]
[= contents <syshistdefs-1 compl create-bp fixf fixfiles fixl
inparent loc-compare makinparent makinparent-def nexbp
read-entity-file readent-cur readent-save savestate-file
symbarg1 writesave>]

@Comment
This section contains actions and functions for saving to, and
retrieving from a history of versions of entityfiles within
one individual.

@Development-status
No remaining things to do in this section.

@Debug-status
No known bugs in this section.

------------------------------------------------------------
-- compl

[= type instant-action]
[= latest-reset bp-0002]

[= comment "
   Compare current version of file with latest saved one. Write
   a report but do not update any properties. See def of function
   loc-compare for more details."]

(definstact 'compl '()
   '(progn
```

```
        (loc-compare (get *curactor* 'curloc) nil)
           )
   ''(seq& nil) )

@Remark
An instant-action is a kind of action that completes execution
within one cycle of the Leonardo executive, and such that even
a sequence of several instant-actions can be performed within
one such cycle. They are suitable as commands from the user, for
immediate execution.
```

```
------------------------------------------------------------
-- create-bp

[= type lisp-function]
[= latest-reset bp-0001]

[= comment "
   Create and register a new breakpoint in the present individual.
   Return it as a value."]

(defun create-bp ()
   (let* ((j (+ 1 (or (get 'latest-bp-nr 'value) -1)))
          (bp (intern (concatenate 'string "bp-" (threeplace j))))
          (bpl (cadr (get 'syshist 'contents))) )
         (make-new-dir (concatenate 'string *myself*
              "Savestate\\" (string bp) ))
         (setf (get 'latest-bp-nr 'value) j)
         (setf (get 'latest-bp 'value) bp)
         ;; ought to do this by setglob or what-it's-called
         (setf (get bp 'type) 'breakpoint)
         (setf (get bp 'date-time)(get 'curchronicle 'value))
                ;; more information should be added here
         (or (member bp bpl)(nconc bpl (list bp)))
         (writeloc 'syshist)
         bp
     ))

------------------------------------------------------------
-- fixf

[= type instant-action]
[= latest-reset bp-0001]

[= comment "
   See comment for function fixfiles."]

(definstact 'fixf '(loclist)
   '(fixfiles loclist)
   '(tkread (concatenate 'string "<" (read-line t) ">")) )

------------------------------------------------------------
-- fixfiles

[= type lisp-function]
[= latest-reset bp-0001]

[= comment "
   This function is used when a new breakpoint needs to be set up
   and used. It first obtains the new breakpoint number.
   Then it goes over the list of entityfiles given as argument
   and identifies which of them need to be rewritten into the new
   saved file, by checking their current contents againt their
   currently saved contents. Changes are noted on the entities and
   on the entire file. If a change, then the file is fixed in the
```

```
   new checkpoint."]

(defun fixfiles (loclist)
  (let*  ( (nbp (create-bp))
           (lfiles (cdadr (get loclist 'contents)))
           (files (mapcar (function symbarg1) lfiles))
           (res) )
          (setf (get nbp 'allfiles)(list 'seq& files))
```

```
         (dolist (loc files)
                 (loc-compare loc nbp)
                 (cond ((equal nbp (get loc 'latest-reset-entity))
                    ;; some changes were detected by loc-compare
                        (setq res (nconc res (list loc)))
                        (writeloc loc)
                        (writesave loc) )))
         (setf (get nbp 'changefiles)(list 'seq& res))
         (setf (get nbp 'date-time)(get 'curchronicle 'value))
            ;; should be replaced by a correct date-time record
         (writeloc 'syshist)
         nbp  ))

------------------------------------------------------------
-- fixl

[= type instant-action]
[= latest-reset bp-0001]

[= comment "
   Save the contents of the current entityfile (curloc) at
   the present bp generation. Save the information in it itself
   that it has been saved at this bp."]

(definstact 'fixl nil
   '(let ((loc (get *curactor* 'curloc)))
       (setf (get loc 'latest-reset-entity)
             (get 'latest-bp 'value) )
       (writeloc loc)
       (writesave loc)
           )
   ''(seq& nil) )

------------------------------------------------------------
-- inparent:

[= type entity-constructor]
[= latest-reset bp-0001]

(setf (get '|inparent:| 'symbfun)
    #'(lambda (x)
        (let ((j (intern (concatenate 'string
                     "(inparent: " (string x) ")" )) ))
             (setf (get j 'type)(get x 'type))
              j )))

------------------------------------------------------------
-- loc-compare

[= type lisp-function]
[= latest-reset bp-0001]

[= comment "
   This function compares the current state of a location with the
```

```
   most recently saved one. If the second argument is nil then it
   only makes the comparison and reports on its findings, but
   does not assign any properties for it. If the second argument
   is a bp (and therefore not nil) then it is intended as the
   future bp at which recent changes will be saved. In this case
   it saves information both ways about this (forthcoming) saveop."]

(defun loc-compare (loc nbp)
   (let ((curents (readent-cur loc))
         (save-ents (readent-save loc))
         (changes) )
```

```
        (cond
           ((not (equal curents save-ents))
            (setq changes t)
            (if nbp (setf (get loc 'latest-reset) nbp))
            (princ "Change of entity-lists:")(terpri)
            (princ "From:  ")(princ save-ents)(terpri)
            (princ "To:    ")(princ curents)(terpri)
            (terpri) ))
        (dolist (x save-ents)
            (cond ((not (member x curents))
                   (setq changes t)
                   (if nbp (setf (get x 'latest-reset) nbp))
                   (prin1 x)(princ " removed")(terpri) )
                  ((not (equal (get x 'curdef)(get x 'saved-def)))
                   (setq changes t)
                   (if nbp (setf (get x 'latest-reset) nbp))
                   (prin1 x)(princ " changed")(terpri) )
                ))
        (dolist (x curents)
            (cond ((not (member x save-ents))
                   (setq changes t)
                   (if nbp (setf (get x 'latest-reset) nbp))
                   (prin1 x)(princ " added")(terpri) )
                ))
        (cond ((and nbp changes)
               (setf (get loc 'latest-reset-entity) nbp) ))
           ))

------------------------------------------------------------
-- makinparent

[= type lisp-function]
[= latest-reset bp-0001]

(defun makinparent (x)(intern (concatenate 'string
     "(inparent: " (string x) ")" )))

------------------------------------------------------------
-- makinparent-def

[= type lisp-function]
[= latest-reset bp-0001]

(defun makinparent-def (x)
   (let ((j (makinparent x)))
        (setf (get j 'symbexpr) (list '|inparent:| x))
        j ))

------------------------------------------------------------
-- nexbp

[= type instant-action]
[= latest-reset bp-0001]
```

```
[= comment "
   Instant action corresponding to the lisp function for
   creating a new breakpoint."]

(definstact 'nexbp '()
   '(create-bp)
   ''(seq& nil) )

------------------------------------------------------------

/transcript interrupted at this point/
```