Decision Support Components on the Internet [1]

Manfred A. Jeusfeld (*), Tung X. Bui (+)

*) RWTH Aachen, Informatik V, Ahornstr. 55, 52056 Aachen, Germany
jeusfeld@informatik.rwth-aachen.de

+) Hong Kong University of Science &Technology, ISMT,
Clear Water Bay Rd., KLN, Hong Kong, tbui@usthk.ust.hk

*Abstract***.** Operational software like online transaction processing systems still take the lion's share of business information systems. Though many Decision Support Systems (DSS) were developed, they failed to become mainstream products. The reason is lack of knowledge of availability, applicability, and interoperability. In this paper, we propose to exploit the vast resource of the Internet to give DSS a second chance. Contrasting to other approaches, our strategy is bottom-up, i.e., users and developers of DSS's cooperate in weaving a web of distributed DSS components. Repositories serve for ensuring application-centered consistency and for searching. The method is scaleable in number of processors, application domains, and platforms. A script language for combining DSS components is introduced based on observations from a real life example of a multi-national company.

---

[1] Most of the work reported here was done while the authors were with the Hong Kong University of Science & Technology, Department of Information and Systems Management, Clear Water Bay Rd., KLN., Hong Kong.

1. Introduction

According to a recent survey by PCWeek (1994), the deployment of Decision Support Systems (DSS) across organizations is rather limited. There are at least three reasons that explain this poor distribution of DSS:

- *Difficulty in matching problems to appropriate DSS*: Decision makers often experience difficulties in finding available analytical models appropriate their decision problems.
- *Up-to-date DSS are not easily accessible*: A comprehensive model-base is hard to maintain as models and tools are being continuously updated or introduced.
- *DSS are often too application-specific*: Models have different assumptions about the input/output formats; some of these assumptions are implicit, i.e., not visible in the declared format.

Recently, with the explosion of Internet use, many academic institutions have started making their DSS components available to potentially interested decisions makers. Bhargava et al. [95;1] cite three cite three factors contributing to this trend. They argue that, due to the currently limited size of the market for decision support technologies, there is a *market potential* for DSS deployment on the World Wide Web (WWW). More important, DSS availability on the WWW could eliminate, or at least reduce, the costs and problems related to the issue of "re-inventing the wheel" and version management. Goul et al. [95] view DSS deployment on the Internet as an innovative means to (i) help discover opportunities for using newly developed DSS, (ii) validate DSS, including continuous feedback on their usefulness and performance, and (iii) promote wider dissemination of DSS via training.

From an operational point of view, the goal is to match the demand and the service of DSS with much lower overhead costs than those of a more conventional method of DSS distribution. Ideally, a decision maker should be able to combine suitable DSS components in the same way that he/she builds and uses functions in a spreadsheet, without having to be concerned with installation of the components in his /her computer environment. An Internet-based usable system must provide answers to the following questions:

- How can a complex decision problem be planned using distributed DSS components as an Internet-based approach to model integration? Our answer is a script language with a graphical and a textual representation.
- How can users access appropriate DSS components dispersed in the potentially unlimited Internet search space? Here we use the idea of Uniform Resource Locators (URL) known from the World Wide Web. Data, scripts, models, and DSS programs are all indentifiable by their URL.
- How can data objects be transformed into the format expected by a DSS component? We propose a version of the uniform data exchange OEM which includes type information with the complex DSS data. This minimizes the handshaking between sender and receiver of DSS data and allows to store a data item independently from ist originator.

- How can the Internet traffic be minimized when data/script objects are at distant nodes? Here, the simplicity of our script language will allow us to apply various optimizations.

Our approach is characterized by a lack of central control. DSS components (data, models, scripts, programs) are created and maintained in an distributed environment. Thus, the sum of all available DSS components may not have a consistent meaning or behavior. Therefore, we propose repository systems mediating between DSS providers and users. These repository systems define which components may be combined and allow the users to browse in an application-specific search space of components.

A Decision Support Assistant (DSA) was proposed by [Felter 89] for local area networks (LAN). It provides an intelligent assistant that helps a decision maker to invoke DSS components from a model base. Felter studied interference of concurrent tool invocations and noticed the limited processing time resources. In our approach, the control of the DSS components is more complex than that in a LAN. The main difference is the degree of distribution. In Felter's approach, all DSS components are controlled by a central DSA. In ours, a federated environment makes a central control both unenforceable and undesirable. In that regard, we depart from the 'brokers-agent' concept proposed by [Bhargava et al. 95;2]. Potential DSS users do not need to buy the service of a broker to make use of distributed computational resources on the Internet.

To allow distributed management of all resources (DSS script and data objects), we present two design principles: A *uniform data representation* as a prerequisite for exchanging information between heterogeneous systems; and *uniform naming* for identifying input/output data as well as the DSS components that participate in a decision process. In addition, we advocate the implementation of repository systems to provide DSS users with the knowledge to design distributed decision support scripts, and to aid the installation of DSS components across the Internet. As a whole, our approach is germane to the framework of the "enterprise information bus" by [Ba et al. 95], yet avoiding the requirement of having a central agent.

A Corporate-Wide Financial Application Example. To illustrate the approach set forth in this study, a case study adapted from real-life application is presented. Rubber International (RI) is a multinational company with its headquarters located in North America. With annual sales of Can$ 1.5 billion, RI has its operation worldwide. With a variety of products offered, RI seeks to constantly monitor its markets regionally. To align with its geographically distributed sales, RI information systems architecture is also distributed with major computing hubs located in Montreal, Canada, and Strassbourg, France, as well as various regional computing centers, e.g., in Fribourg, Switzerland.

As part of its decision support and executive information systems, RI has developed a financial planning system. The purpose of the system is to help management establish

- yearly and 5-year budgeting
- short term and long term strategic planning
- cash flow management
- routine and ad hoc data analysis and reporting

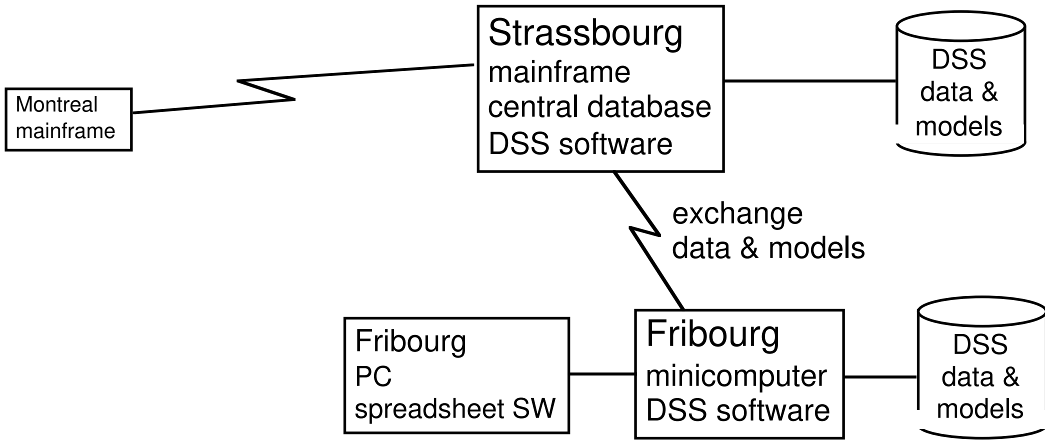

Fig. 1: Architecture of an existing federated DSS

Figure 1 shows an excerpt from the current federated DSS architecture of RI. The European headquarters in Strassbourg maintains a financial data and models for the company's European operation. Regional centers, like Fribourg in Switzerland, contribute to the headquarters data from their operation but also require information from it for local decision making. Some financial data and models are kept locally at the regional centers. Both the headquarters and the regional computer centers provide DSS components for evaluating the financial models on the financial data. Spreadsheets programs running on personal computers serve for visualizing results of the analysis. The networking of RI's federated DSS architecture has been implemented by leased data communication lines and dedicated communication protocols. The location of the financial data and models were determined in group meetings between the headquarters staff and the regional centers staff.

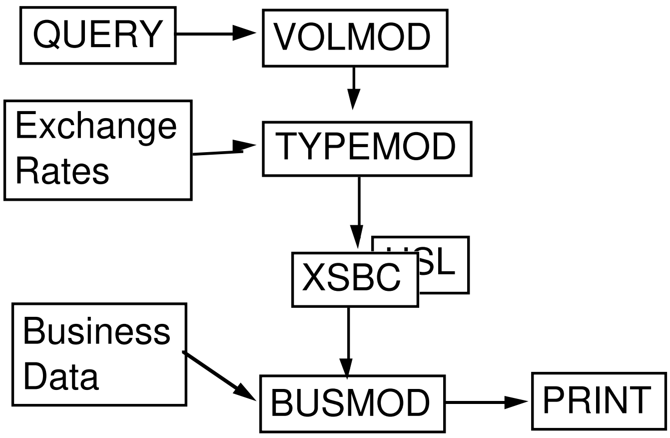

Fig. 2: A script for financial analysis (from RI)

Scripts for computing an analysis were specified by semi-formal data flow diagrams (see Fig. 2) and then implemented by a 4GL language. The boxes on the left-hand side represent financial data, the boxes in the middle are financial models that define how to extract relevant information from the data, and finally the box 'PRINT' specifies how the output is treated. RI also has examples of scripts with loops, e.g. doing the same analysis for all month of a fiscal year.

The example shows that knowledge about the DSS is spread in heterogeneous documents (data, models, scripts). Interoperability, i.e. the ability to pass data between systems without information loss, is ensured by using the same software packages and by dedicated transformation routines (wrappers). Scripts have a Janus nature: on the one side they can be viewed as a high level programming language, on the other side they define a semantic relationship between data and models.

2. Overview of the Proposed Federated System

We use the setting of RI as a running example for our method of integrating DSS components on the Internet. Our goal is to provide a scaleable method where data, models and processors may be located anywhere on the network. Users (decision makers, analysts, DSS software providers, etc.) of the proposed federated architecture should be able to exchange information after an agreement on location, representation and interpretation of the information has been reached. The users may be in different departments of the same company but may also be from different cooperating companies.

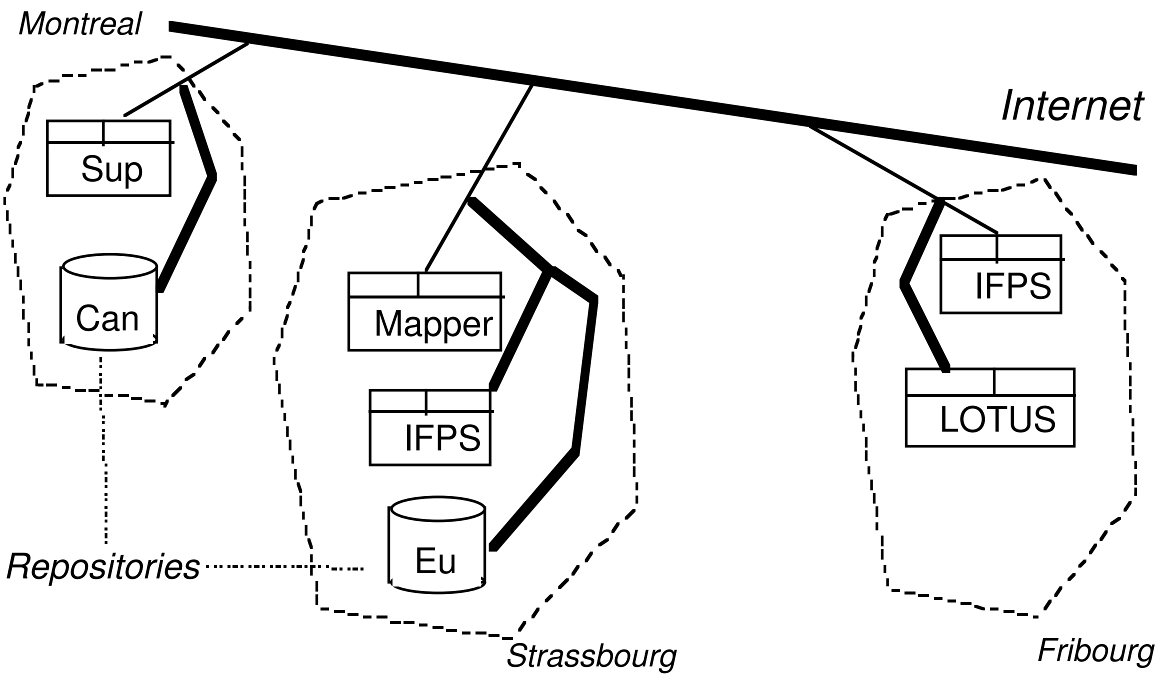

Fig. 3: Overall architecture of the proposed federated DSS

Figure 3 shows a simplified view of a network of DSS processors and repository systems (labeled 'Can' and 'Eu' in the Figure) connected by the Internet (visualized by the thick lines). The rectangular nodes represent DSS processors, i.e., programs that process information relevant for some decision. The cylindrical nodes (here 'Can' and 'Eu') are examples of repository systems. The repositories constitute the foundation of the overall system in that they serve as the main information resource to support discovery and composition of Internet-based DSS components. The repositories can be implemented by a standard database application program, or by knowledge-based systems, each dedicated to a specific application domain or purpose (e.g., repository of financial applications, repository for statistical procedures, etc.).

Each repository creates its own name space, i.e., the set of stored uniform names for script and data objects. Specialized name spaces are justified for several reasons. First, different groups of decision makers are interested in different DSS components. Second, and more importantly, distributed repositories create a higher degree of security because script and data objects are known only to those users who have access to the repository.

One of our goals is to help DSS providers promote the access and use of their systems. The overall system can be continually updated by installing new DSS processors:

1. A user (system administrator or even decision maker) browses a repository, looking for DSS processors suitable to her problem definition, and selects one.
2. The repository is queried for the installation procedure of the selected DSS processor(s).
3. The installation procedure is downloaded to the local system and executed, provided the features of the procedure conforms to the local system configuration.
4. At the end of a successful installation, script objects for calling the new copy of the DSS processor have to be inserted in the local repository (uniform name, input/output types, etc.).

As more and more copies of the same DSS package are spread over the network, redundancy can be exploited for optimizing the load balance of the system (see section 5). A DSS provider 'publishes' a new DSS component by inserting its features in the repository system known to him. This can include a test installation at the local site of the DSS provider to help potential users test the component(s).

3. Interoperability Techniques

Let’s consider the following generic scenario. A DSS component P1 produces a data object O1 which is later used by DSS component P2 as input. The DSS components are autonomously developed and the data exchange via O1 may not have been planned at the time P1 and P2 were designed. The traditional approach is to store data in a (central) database for later processing (e.g., [Sprague 1989]). For this purpose, the schema of the database must be known in advance and is relatively fixed. In our approach, data as well as script objects are assumed to be distributed and no common type system is enforced.

Remember two principles for distributed processing: first, data can be packaged into a transferable exchange format similar to IDL [Nestor et al. 90]; second, Uniform Resource Locators (URL) [Barners-Lee 95] can be used to identify DSS components installed on the Internet.

Uniform Data Representation. We regard any data as a string of bytes to be received and interpreted by the DSS component which processes it. This option requires external knowledge about the kind of data, and is therefore not suitable in distributed environments where data and script objects evolve without central control.

There are several similar approaches from the areas of distributed systems, heterogeneous databases, and distributed artificial intelligence coping with the problem of data exchange. We adopt the OEM format [Papakonstantinou et al. 95] which incorporates the type information within each data object, i.e., the data structure can be retrieved from the object representation. Objects are represented as terms (n, t, v) where 'n' is the name of the object, 't' is its type and 'v' is its value. The value is either a constant of a base type or a term. As an example, consider the definition of a vector of currency units in Figure 4. The building principle of OEM objects are triplets '(instance-name, type, value)'. The first component identifies the object, the second is the name of the type of the object, and the last is the value. OEM objects may be arbitrarily nested. Nested objects can be identified by a path expression, e.g., 'Selling-Data-1995.Q-2' identifies the Canadian dollar amount 948 (in thousand). Such a representation makes a data object independent from type systems of programming languages: it simply carries its type definition with it. A system that reads such a term can extract the type information and match it against its own type system.

```
(Selling-Data-1995, Selling-Vector,
     (     (Q-1, K-CAN$, 1033),
           (Q-2, K-CAN$, 948),
           (Q-3, K-CAN$, 1018),
           (Q-4, K-CAN$, 316)    ))
```

Fig. 4: OEM object of type matrix

Uniform Naming. The input of DSS scripts are data objects that may reside on one or more (remote) locations. Therefore, both data objects and script objects must be identifiable across system boundaries. Uniform resource locators (URL) [WWW-Consortium 95] serve this purpose. The general format is

```
protocol://node-address/path
```

The component 'protocol' specifies the communication protocol used to transfer data or to activate a remote process. We assume the widely-used protocol 'http' in the following. The 'node-address' uniquely identifies the computer system which hosts the requested data or script object. Finally, the 'path' identifies the object within the name space of the local computer system. For example, the URL

```
http://strassbourg.rubber.fr/finance/
          Selling-Data-1995.oem
```

uniquely identifies location of the data object Selling-Data-1995 in the file system of the computer 'strassbourg' of organization 'rubber' in France. Via the transfer protocol 'http', it is technically accessible from any node in the network in which 'strassbourg' participates. We assume that any *persistent data object* , i.e., a data object that exists beyond the processing time of the script objects that produced or consumed it, is represented in a file with extension 'oem' indicating that its content is in OEM format.

A call to a DSS processor is also be named via URLs. The format resembles a procedure call with input parameters referenced by their URLs. For example, the URL

```
http://fribourg.rubber.ch/bin/IFPS?call=http://
strassbourg.rubber.fr/scripts/Earning-Forecast-1995.oem
```

invokes the DSS processor IFPS (actually a financial analysis package) at Fribourg to evaluate on the input object Earning-Forecast-1995.oem located in Strassbourg. As specified in the HTTP protocol, such an URL names the result of the evaluation. This is exploited in the definition of script objects below.

Figure 5 shows a script object where all objects reside on different nodes in the Internet. The *script object* Earning-Forecast-1995.oem encodes the call of the DSS processor IFPS at Fribourg, and the arguments of the call which are either URLs of data objects or calls of component scripts. Argument arg-1 of Earning-Forecast-1995 is financial model BUSMOD, arg-2 is Selling-Data-1995 from Strassbourg, and arg-3 is the result of a nested call to another DSS processor IFPS at Strassbourg.

```
(Earning-Forecast-1995, po-call,
   ((po-node,url,http://fribourg.rubber.ch/bin/IFPS),
    (arg-1,url,
     http://fribourg.rubber.ch/local/BUSMOD.ifps),
    (arg-2,url,
     http://strassbourg.rubber.fr/finance/Selling-Data-
1995.oem),
    (arg-3,po-call,
      ((po-node,url,http://strassbourg.rubber.fr/bin/IFPS),
      (arg-
1,url,http://strassbourg.rubber.fr/local/XSBC.ifps),
      (arg-2,url,
       http://strassbourg.rubber.fr/local/Exchange-Rate-
1995.oem))
)))
```

Fig. 5: OEM representation of a script object

The evaluation of a script object is explained in section 5. For the moment, it is sufficient to note that the script object encodes a data flow diagram as depicted in Figure 6. Also note that a script object is in OEM format like the data objects! Therefore, it may be the output of a compiler script. Furthermore, it can be classified in the repository presented in section 4.

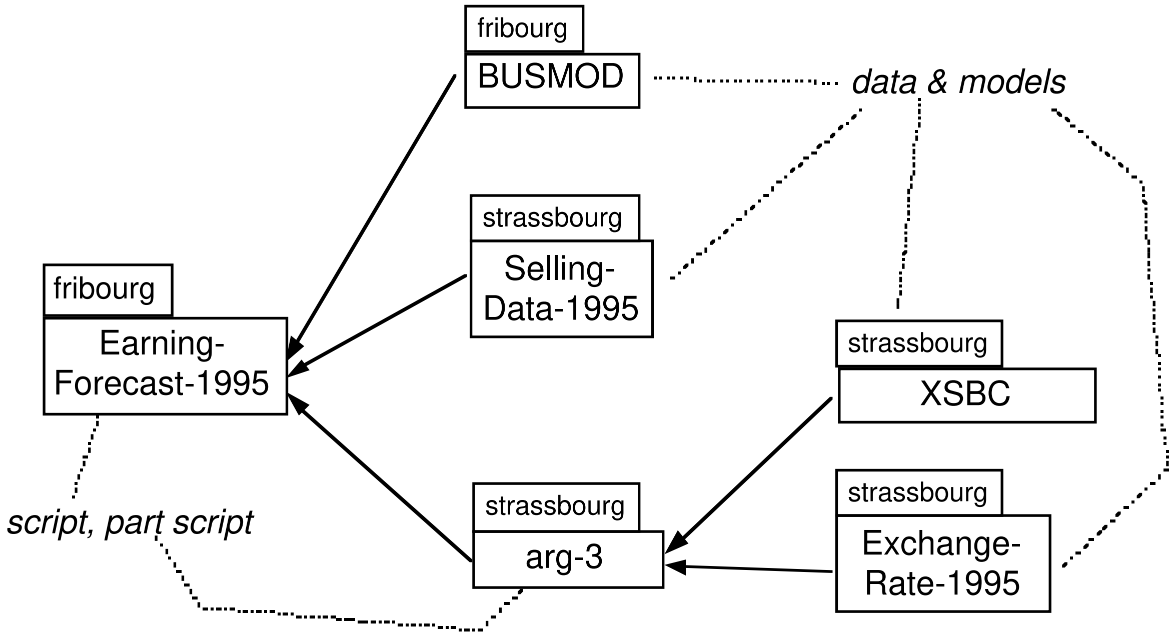

Fig. 6: Diagram denoting a distributed DSS script

The URL conventions are sufficient to invoke a remote DSS component with local or remote input objects and to receive the result in as a single object in OEM format. Legacy DSS components can be integrated by a transducer [Genesereth & Ketchpel 94] which mediates between the network and the DSS component. The transducer accepts the call in the above syntax, transforms the (input) data into the internal representation for the legacy DSS component and then invokes it.

The proposed approach is well-suited for DSS components with no user interaction, i.e., those that are linked with other DSS components. The design of local user interface should not impose principal obstacles (i.e., at the client site in a client/server framework). An elegant way is to install a specialized DSS processor on the computer system of the user. This processor accepts requests from the remote DSS component via the URL call mechanism shown above. Locally, it gathers the input from the user by an appropriate input device.

4. The Repository

Uniform data representation and naming described above provide the basic framework for communication between distributed DSS components. We propose a repository which stores conceptual information about data objects as well as script objects and answers queries based on classification hierarchies.

Persistent Data Objects in the Repository. Each data object is in OEM format. Thus, its type can be extracted from its representation and be used for classifying it in the repository. We use the statement ‘OBJECT x IN T’ to express that an object ‘x’ is an instance (element) of type ‘T’. The type is represented as a term over the type constructors and base types (given by their name). For example, the most specific type of Selling-Data-

1995 is Selling-Vector(K-CAN$,K-CAN$,K-CAN$,K-CAN$). Consequently, the data object is represented in the repository as:

```
OBJECT http://strassbourg.rubber.fr/finance/
              Selling-Data-1990.oem
IN Selling-Vector(K-CAN$,K-CAN$,K-CAN$,K-CAN$)
```

Membership of the data object to more generic types like Selling-Vector(Int,Int,Int,Int) can be derived provided Int is a more generic type than K-CAN$. We express such generalization in the repository by a statement ‘K-CAN$ ISA Int’. Further classifications can be obtained when additional attributes about the data object are provided. Such attributes are highly domain-specific, especially when data objects come from different organizations. Therefore, it is difficult to prescribe a set of attributes to be used. Book-keeping features like 'creator', 'creation-date', 'valid-thru', and 'comment' are reasonable first choices. Semantic information explaining the meaning of a data object, its problem class and applicable models is more useful for data classification but requires a common terminology among the users of the repository (which is beyond the scope of this paper). In our running example, the attributes could be as follows:

```
K-CAN$ ISA Int
...
OBJECT http://strassbourg.rubber.fr/finance/
              Selling-Data-1995.oem
IN Selling-Vector(K-CAN$,K-CAN$,K-CAN$,K-CAN$)
WITH ATTRIBUTE
     creator: "Francois Nome";
     creation-date: "4-Nov-1995";
     valid-thru: "31-Dec-1995";
     remark-1: "entries correspond to quarterly selling
                 of Rubber-Europe";
END
```

DSS-Scripts are special data objects that describe a computation for a certain decision problem. In the example below, the script Earning-Forecast-1995 is defined with two arguments of type IFPS-Model (a plain text) or Selling-Vector, respectively. The result is expected to be of type matrix(Int). Note that the input and output signature can be used for searching for data objects matching a script’s signature and for checking type conformance.

```
OBJECT http://strassbourg.rubber.fr/scripts/Earning-
Forecast-1995.oem
IN DSS-Script
WITH ATTRIBUTE
     arg-1: IFPS-Model;
     arg-2: Selling-Vector({Int});
     arg-3: Exchange-Vector({pair(Real,Real)})
     result: matrix(Int)
     preferred-processor:
http://fribourg.rubber.ch/bin/IFPS
END
```

The processors of scripts (attribute preferred-processor in the example above) are DSS software packages installed on some computer in the network. It may well be that the same DSS package is installed on several nodes. Then, there is a choice for the evaluation strategy of the script. In the repository, we store a statement like

```
OBJECT http://fribourg.rubber.ch/bin/IFPS
EQUIVALENT-TO
http://strassbourg.rubber.fr/software/bin/IFPS
```

to represent this fact. It is used for the run-time scheduling of the evaluation of a task.

Some scripts may have the only task to filter, transform or display some data objects. They constitute no special case. For example, the result of Earning-Forecast-1995 may be transformed into a spreadsheet format and then displayed by a standard spreadsheet package.

5. Implementation Steps

To implement the proposed approach, two principle steps are required. First, the DSS components have to be connected to the Internet, and second , they must be updated in the repository for identification.

Connecting DSS Components to the Network. Our approach relies on standard protocols and interfaces. Therefore, setting up a running system does only require a fair amount of programming. First, the DSS processors must be accessible from distant nodes. This can be achieved by installing a so-called HTTPD server [WWW-Consortium 95] on the local node. The server has the basic function of accepting calls from the Internet and returning some result (in our case data objects). It can be configured to transfer all calls with the right URL to the local DSS component. Second, a local DSS component must be able to retrieve the arguments of a call (as encoded in the script object), i.e., to call itself remote HTTPD servers. The standard-compliant library 'libwww' for this script is freely available from [WWW-Consortium 95]. Figure 7 shows a schematic view of a DSS component linked to the WWW.

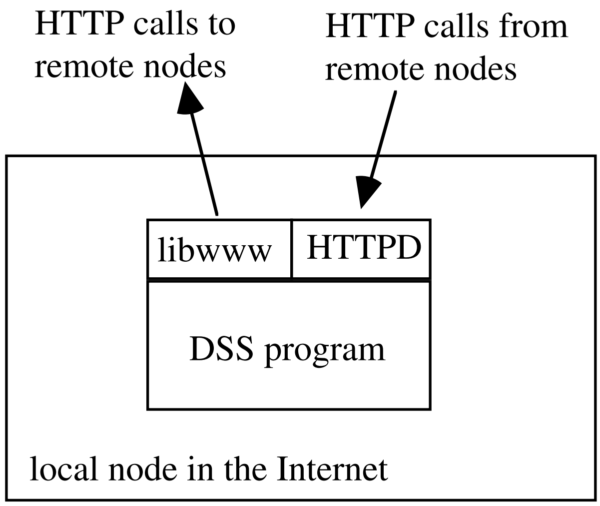

Fig. 7: Schematic architecture of a DSS processor linked to WWW

This architecture allows a user to activate a DSS processor from a remote node with a certain script object. As third and final requirement for the connection, the arguments and the result of a call to a script are expected to be in OEM format. Therefore, the DSS processor must be able to read/write such a format and transfer the values to and from its internal format. The effort for this function is minimal once the OEM format is known. A script is invoked by submitting a call like

```
http://fribourg.rubber.ch
      /bin/IFPS?call=http://strassbourg.rubber.fr
      /scripts/Earning-Forecast-1995.oem
```

to the HTTPD server of a suitable processor. The DSS processor then fetches the script object (here Earning-Forecast-1995.oem) through its libwww interface. Argument descriptions in the script object which are of type 'po-call', for example arg-3 in Fig. 5, are stored on a local file. e.g. http:// fribourg.rubber.ch/temp/arg-3.oem. Thereby, the local DSS processor can issue the corresponding call, in our running example

```
http://strassbourg.rubber.fr/software/bin/IFPS?
       call=http://fribourg.rubber.ch/temp/arg-3.oem
```

in the same manner it was called itself. The local DSS processor at Fribourg may well deviate from the preferred processor specified in the script and call instead an equivalent processor, e.g. the IFPS package at the local site. This scheduling requires no central control and is guaranteed not to influence the result of the computation. Of course, the DSS processors must be equivalent. Finding equivalent processors is done by a query to a (local or remote) repository.

Searching the Repository. The main functions of the repository system are to store the properties of data and script objects as shown above, and to facilitate searching for information about data and script objects. Three search strategies are discussed below.

Simple search for scripts. Assume a decision maker has a problem encoded by a single input data object and wants to find a suitable collection of DSS components. The solution can be determined by matching the type of the data object with the input argument types of the DSS scripts stored in the repository. The search may be limited by conditions on the features of the DSS script. The search is efficient, more exactly, it is linear in the number of objects in the repository. Indexes on the input argument feature can speed up the average case performance to constant time.

Chain search for scripts. Assume that the repository contains some data transformation scripts P1,..,Pk such that the input argument of P1 matches the input data object and the output for Pi (i=1,...k-1) matches the input of Pi+1. Then, suitable DSS scripts can be obtained by searching such chains and then finding a DSS scripts whose input matches the output of Pk. This search delivers more alternative solutions than the simple search depending on the number of data transformation scripts known to the repository. This search is linear in the number of matches between input and output argument types of DSS components in the repository. In the 'worst' case (i.e., every output type can be transformed to every output type), it is quadratic in the number of DSS components. The quality of the search, and subsequently, the problem solution depend on the completeness of the set of transformation scripts

Interactive script object generation. This is both the most useful and most difficult search. It assumes that the decision maker has more than one data object as problem definition and that some goal objects have to be computed by a complex sequence of DSS scripts. Theoretically, all combinations of chains, each starting with one of the input data objects, are applicable. Indeed, the number of combinations explodes as the number of input data objects and available scripts increases. The problem of finding a suitable script object is similar to scheduling problems. Since the repository has incomplete knowledge about the available script objects (only input/output types plus some features) and about the goal of the decision maker, we propose an interactive approach to construction of the script object. The user incrementally builds the script starting from the input data objects. Simple and chain search are used as auxiliary functions which offer the decision maker a choice of possible component scripts operating on a selected data object (either input data object or output of a script object). After validation the generated script may be included in a repository for later use and re-use (then being component of other scripts).

Implementation Properties. The script objects encode functional expressions on data objects (OEM format). The functions are denoted by the URLs of the DSS processor, and the arguments are either functional expressions or URLs of data objects. We assume the following computational model for DSS processors:

Persistent data objects are never updated after their creation.

That means that the OEM term referred to by an URL is always the same. This property is analogous to the treatment of variables in functional and logic programming languages. It is different from the destructive updates allowed in imperative programming languages. The result of a script object only depends on its input objects. As a consequence, two calls of the same DSS processor will have the same result if and only if the input objects are persistent data objects.

The computational model allows application of several heuristics for optimization of script objects.

- *Lazy evaluation* lets arguments be evaluated only when actually requested. This heuristic reduces the overall load on the network. Its implementation is not trivial because it affects the internal routines of the DSS components.
- *Caching* materializes results of sub-expressions (parts of script objects which are of type 'po-call' for speeding up subsequent calls of the same sub-expression. Caching makes most sense for script objects having only persistent data objects as input since their result never changes.
- *Multi-processing* allows arguments of a script object which are of type 'po-call' to be called in parallel. This strategy is the opposite to lazy evaluation and will speed up response time provided the calls go to distant idle nodes in the network. A precondition for this strategy is that the parallel calls do not interfere with each other. This can be achieved by forbidding updates to persistent data objects.
- *Load balancing* can be enforced to optimize network traffic. Calls to DSS processor (named by their URL in the script object) can be replaced by calls to 'equivalent' script objects located elsewhere. Load balancing can be incorporated before evaluation (compile-time approach) or during evaluation (run-time approach). Since DSS processors may be installed redundantly at any time on any node in the network, the overall system is scaleable to the workload. There is no need of central control. The heuristics can be applied with the (limited) knowledge of the local repository system.

Script objects are hierarchical, i.e., they do not contain loops. The evaluation of a script object will terminate if the evaluation of its components (either task objects or data objects) terminate. We conclude the following property:

> *The evaluation of script objects will always terminate under a sound evaluation strategy (e.g., top-down, depth-first).*

This property allows DSS users not familiar with programming languages to design them without fearing to consume too much resources of the network. Indeed, the resource consumption can be estimated from a script object definition. The total processing time is the sum of the processing times of the component scripts in the script objects, i.e., it is linear in the size of the script object. The processing times of the individual calls to DSS processors are in general indefinite. Response time, i.e., the time between calling a script object and receiving its result, depends on the degree of multi-processing. The minimal response time is the processing time divided by the number of distinct nodes involved in the processing. The actual response time also depends on data dependencies and the network bandwidth. Network costs (measured in the amount of data to be transferred between distinct nodes) is also linear in the size of the script object since in the worst case only as many transfers occur as there are DSS processors referred to in the script object. Network costs are also linear in the size of the input data objects provided that the result of any call to a DSS processor is linear in the size of its inputs. This is a reasonable assumption for DSS components since their purpose is to reduce decision problems on large data sets to a small number of choices. Network costs can be reduced to zero by installing any DSS processor referred to in the script object on the local node of the Internet. This is an extreme case of load balancing whose expenses are strictly those from local resources.

Integrity and Control. On first sight, the distributed approach on the Internet bears the danger of uncontrolled access to models which where never meant to be combined. The reader should however note, that the repositories are specifically used to define a name space of meaningful data and scripts objects for a certain group of users. In the extreme case, a repository can just cover the objects on a local computer network. On the other hand, even small organizations have worldwide connections and will need the techniques for global decision support. The repository for such an organization may and should be restricted to cover those distributed objects important for its business.

As seen for the application of electronic data interchange (EDI), distribution of information processing requires a clear understanding of the business processes of the participating organizations. We assume that this understanding has been reached *prior* to the exchange of models and software. The 'global' DSS, i.e. the sum of all data and script objects on the Internet, can never assumed to be consistent. Instead it is the resource from which consistent subsets (maintained in the decentralized repositories) can be extracted. This global resource will increase in value when more and more data transformation scripts are published in the repositories. They are in fact the tools for enabling interoperability.

6. Final Remarks

A combination of three elements describe our approach. Uniform naming of script and data objects allows the programs and data to be located anywhere in the Internet. A uniform data format, OEM, allows data exchange independent from the implementation language of the DSS components. Finally, decentralized repositories support the design of distributed scripts and the evolution of the overall system by providing users with semantic information.

The proposed approach relaxes the necessity of having central control of the overall system. While the lack of a central clearinghouse creates some drawbacks, e.g., problem formulation, book-keeping issues, our approach empowers an unlimited number of decision makers (users) and DSS providers to participate in the growth of the overall system. It also provides more flexibility in terms of DSS proliferation, diffusion and use. Utilizing widely accepted protocols for data exchange keeps the implementation costs reasonably low. Also, the effects of low bandwidths of existing Internet traffic can be reduced by duplicating DSS processors on the nodes (computers) where the data reside, and a large number of nodes can be exploited via multi-processing.

Future efforts should concentrate on more sophisticated search facilities for the repositories and quality control of retrieved DSS scripts. Two other research issues are the cooperation among several decision makers and guidance in long-term decision processes. The approach presented in this paper has only marginal support for several decision makers cooperating in a decision script. To extend our approach to real cooperation, one has to embed the individual script objects into complex script or process languages [Ellis & Jarke 92]. Then, loops for scripts are likely to become an issue. We are currently investigating such limited forms of such loop concepts.

References

[Ba et al. 95] S. Ba, Kalakota, A.B. Whinston, "Executable Documents as the Basis for DSS", Proc. 3rd Intl. Conf. on DSS Hong Kong, 22-23 June 1995, pp.373-380.

[Barners-Lee 95] T. Barners-Lee, Uniform Resource Locators, Document http://www.w3.org/hypertext/WWW/Addressing/URL/URL_TOC.html, 1995.

[Bhargava et al. 95;1] H. Bhargava, A. King, D. McQuay, "DecisionNet: An Architecture for Modeling and Decision Support over the World Wide Web", Proc. 3rd Intl. Conf. on DSS Hong Kong, 22-23 June 1995, pp.499-505.

[Bhargava et al. 95;2] H. Bhargava, R. Krishnan, D. Kaplan, "On Generalized Access to a WWW-based Network of Decision Support Services", Proc. 3rd Intl. Conf. on DSS Hong Kong, 22-23 June 1995, pp. 507-515.

[Ellis & Jarke 92] C. A. Ellis, M. Jarke (eds.), Distributed Cooperation in Integrated Information Systems, Proc. 3rd Intl. Workshop on Intelligent & Cooperative Information Systems, Report AIB-92-18, RWTH Aachen, Germany, 1992.

[Felter 89] R. Felter, Decision Support Assistant, Deutscher Universitätsverlag, Wiesbaden, Germany, 1989.

[Genesereth & Ketchpel 94] M.R. Genesereth, S. P. Ketchpel, "Software Agents", Communications of the ACM, 37, 7, 1994, pp. 48-53 & 147.

[Goul et al. 95] M. Goul, A. Philippakis, M. Kiang, D. Fernandes, B. Otondo, "Towards a Client/Server Open-DSS Protocol Suite for Automating DSS Deployment on the World Wide Web", Proc. 3rd Intl. Conf. on DSS Hong Kong, 22-23 June 1995, pp. 517-528.

[Nestor et al. 90] J.R. Nestor, J.M. Newcomer, P. Gianni, D.L. Stone: IDL - The Language and its Implementation, Prentice Hall, 1990.

[Papakonstantinou et al. 95] Y. Papakonstantinou, H. Garcia-Molina, J. Widom, "Object exchange across heterogeneous information sources", Proc. 11th Intl. Conf. on Data Engineering, Taipei, Taiwan, 6-10 March 1995, pp. 251-260.

[Rasmussen et al. 91] J. Rasmussen, B. Brehmer, J. Leplat, Distributed Decision Making - Cognitive Models for Cooperative Work, John Wiley & Sons, 1991.

[WWW-Consortium 95] WWW-Consortium, CERN Httpd, Document http://www.w3.org/hypertext/WWW/Daemon/Status.html, 1995.