

Asset-based system and software system development – A frame-based approach

Jahangir Karimi^a, M. K. Zand^{b,*}

^aUniversity of Colorado at Denver, Denver, Colorado, USA

^bDepartment of Computer Science, University of Nebraska at Omaha, Omaha, NE 68182, USA

Received 15 March 1996; received in revised form 9 February 1998; accepted 13 February 1998

Abstract

Previous works on software reusability has shown that an asset-based systems development strategy to software reuse is effective in identifying reusable design components. This strategy requires moving away from traditional and stand-alone applications development toward an integrated approach. This paper will review previous work on using frames for knowledge representation and will show how an implementation strategy based on the frame-based representation can be useful as a possible implementation approach for the asset-based systems development strategy. © 1998 Elsevier Science B.V.

Keywords: Asset-based systems development strategy; Software reuse; Frame-based implementation; Software design

1. Introduction

In today's marketplace, a competitive software development shop is one that can reduce product delivery time, increase diversity of the product, enhance interoperability of its products, and conform to standardization of components. In the past decade, software reuse has gradually started coming into the mainstream of research and practice as a viable subfield of software engineering. There have been many papers and articles in technical, professional, as well as trade publications dealing with the issues and prospects, the pros and cons, and the incentives and impediments to software reuse [1].

Software reuse is the reapplication of a variety of components of existing systems to a new and similar system. Reuse practice exhibits significant break-through, far more than other on-going activities, to enhance the software development process and to restructure not only the process of software construction but also the actual software development departments. To accommodate for and to exploit software reuse, the management and organization of these departments are to be restructured not just locally and in isolation, but also in the context of the entire organization [1,2–5]. Furthermore, existence of business modeling and domain-specific systems are considered to be of prime factors in success of development reuse project. About 85% of

the reuse reported at IBM have come from domain-specific libraries [6]. Isoda emphasizes the selection of a specific appropriate domain, followed by performing domain analysis on that domain to develop specific domain modules, as one of the primary factors in the success of a reuse project. One of the major problems of the NTT project was the inappropriateness of the target domain. The REBOOT approach too emphasizes the domain-oriented method as a 'secure way' to produce reusable components [1,7].

A lack of a clear strategy has hampered the development of software reuse [2,8]. The traditional approach to software development lacks focus on: (1) planned reuse; (2) system development from an integrated, i.e., organizational rather than stand alone application, perspective; and (3) long-term organizational strategic advantage. Asset-based systems development strategy to software reusability have proposed to address the above mentioned objectives [5,9]. This strategy requires moving away from traditional, stand-alone applications development toward an integrated approach. It translates long-term organizational strategic needs to short term systems development objectives. One study reported the experience of a large bank in implementing an information systems strategy that is based on the concept of reusability [10]. The study reported that, although the reuse strategy described in the bank study was independently arrived at, the reuse strategy used was in essence equivalent to the asset-based systems development strategy proposed in another work [9]. In bank experience, it was

* Corresponding author.

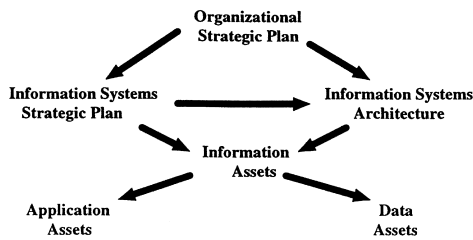


Fig. 1. Asset-based system development strategy.

confirmed that applying the reuse strategy saved the bank over \$1.5 million in development costs in one year alone. In the bank study the reuse strategy was effective in identifying reusable design components. Biggerstaff and Lubars have point out that “...Having a stable, well understood software asset base is crucial to achieving successful software reuse” [11]. Proposing an effective method for implementing these components is still an open area for research.

This paper will focus on proposing an implementation method for the asset-based systems development strategy. First, this strategy is briefly reviewed. The overview of the previous work on using frame for knowledge representation and frame-based reasoning are presented next. This overview will show how frames allow for segmentation of various parts of domain knowledge into sub-units. Finally, this paper will show how an implementation method based on the frame-based representation can be useful for the asset-based systems development strategy.

1.1. Asset-based systems development strategy

As shown in Fig. 1, the asset-based systems development strategy is different from traditional applications-based systems development strategies in many ways; the differences are differences in planning, organization, administration, and control of information resources. This strategy takes into account how the organizational strategic plan must drive the information systems strategic plan which, in turn, determines the information systems architecture components¹. In this article these components are called information assets. The commonality of data process types among applications is the basis for identifying application and data assets. Major advantages of identifying commonality at the organizational level is to prevent multiple development and to minimize maintenance efforts. By minimizing design redundancies of these applications at the organizational level, the assets are constructed and maintained only once (even though they may exist in multiple applications at the implementation level). Therefore, this strategy emphasizes reusability at the design rather than the code level. Based on this strategy, a method is proposed to: (1) identify the scope of integration; (2) build a semantic model of the scope; (3) identify application and task categories; (4) identify application

assets; and (5) identify data assets. A brief description of the asset-based systems development strategy follows.

1.2. Identifying the scope of integration

Within the context of application development, the scope of the integration is defined as a set of applications that are chosen for strategic reasons for composing together. Within a bank for example, these can be high-volume, transaction processing type banking applications (e.g., demand deposit, credit card, installment loan, etc), where a good deal of commonality exist among them. The data and process requirements within the set are used for composing applications together in an integration process. This process can be done at several levels: subsystem (program or module) functional (application), and operational mode (decision support, administrative, or transaction processing). Each scope of integration should contain only one operational mode. For example, operational systems are usually separate from administration systems. For all practical purposes because operational systems represent one mode of operation and administrative systems another mode, they should be separated. The model that results when operational and administrative systems are mixed tends to be very confusing and misleading.

1.3. Building a semantic model

Both a top down and a bottom-up analysis approaches are used in integrating applications within the scope of integration, because employing only one of the approaches may not guarantee accurate results². In this integrated approach, semantic modeling is used to represent structural aspects of business data using objects, attributes, type constructors, and *is-a* relationships [12]. In semantic modeling any concept is first identified as an object and then objects are organized into object-types or classes. For example, Fig. 2 illustrates the financial system of a bank. It shows the relationships among classes and sub-classes within the system. In this example, the financial system of the bank is decomposed according to its internal organization. However, the business of the bank is the concept used to create the semantic model. In order to relate object types, the emphasize is on the functionality of those objects. Relations in semantic model are shown by *is-a*. Further, the relationships between objects or classes are viewed using abstraction mechanisms: generalization, aggregation, or classification [13]³. The goal of semantic modeling is to insure that the scope of the integration is covered; by ensuring that all major objects

² Using a top-down analysis alone may result in disregarding details of functions (processes) and data within those applications. On the other hand, a pure bottom-up analysis approach may overlook some of the major functions and the inherent structure of those functions.

³ In *generalization*, like objects are abstracted by relating them to a higher-level object. This relationship can be characterize as an *is-a/sub-set-of-relationship*. In *aggregation*, an object is related to its sub-components via a *is-part-of-relationship*, and it is modeled based on its properties. In *classification*, specific instances of an object are related to a higher-level object via the *is-instance-of-relationship*.

¹ The information system architecture of an organization is a general model of the desired structure of the organization's information systems. It is usually consists of data, application and network architectures.

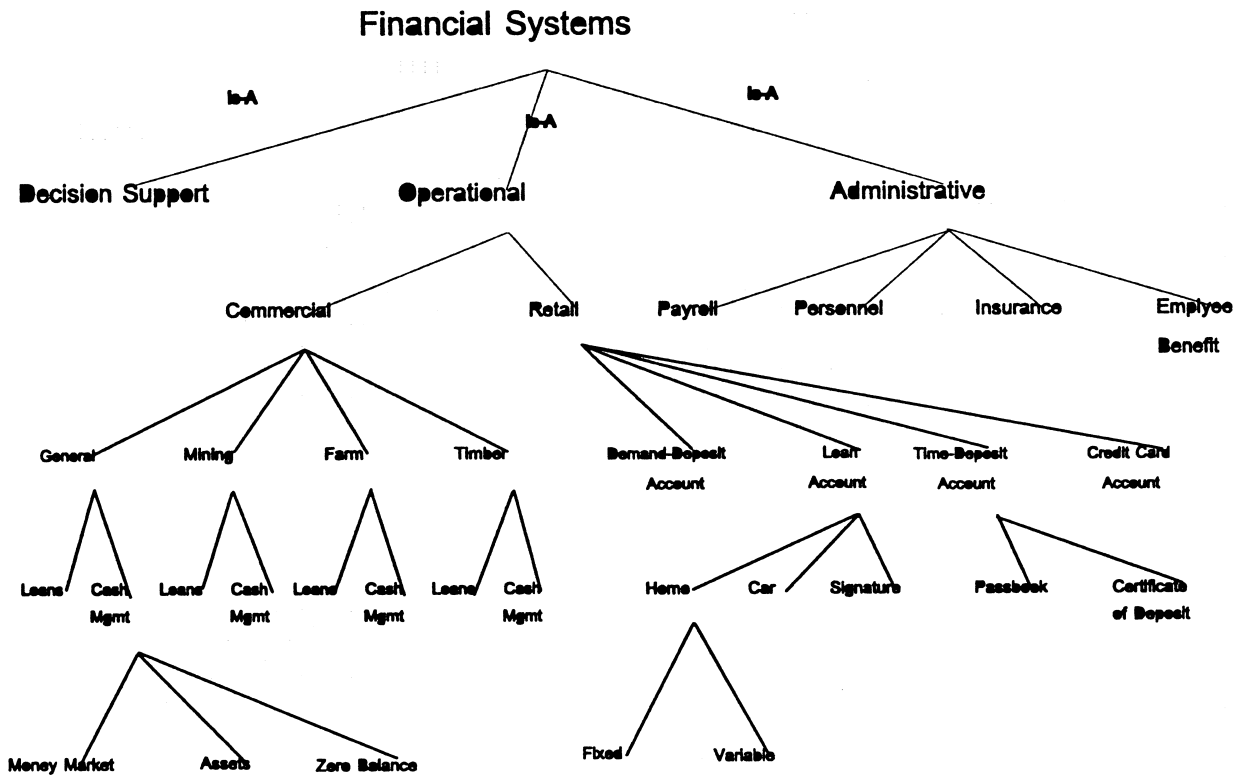


Fig. 2. Generalization abstraction of the financial system of banks.

are represented and properly related to each other. By applying a top-down analysis approach to the system, the base (most abstract) objects are defined first and then the subtypes are defined in a downwards move.

Beside looking at the functions within the scope of integration, the semantic modeling is also done by looking at the data. The entity-relationship diagrams are used to create a global ERD [14]. Here the emphasis is on selecting common entities. As illustrated in Fig. 2, the operational systems are divided into commercial and retail banking.

However, due to common entities shared in both commercial and retail banking, only one ERD is constructed to reflect both aspects of bank business. This global ERD is represented in Fig. 3.

1.4. Identifying tasks and application categories

Primary processes (functions) are identified at the same level of abstraction as those in the global ERD model. The emphasize is on identifying processes that are different⁴. Each

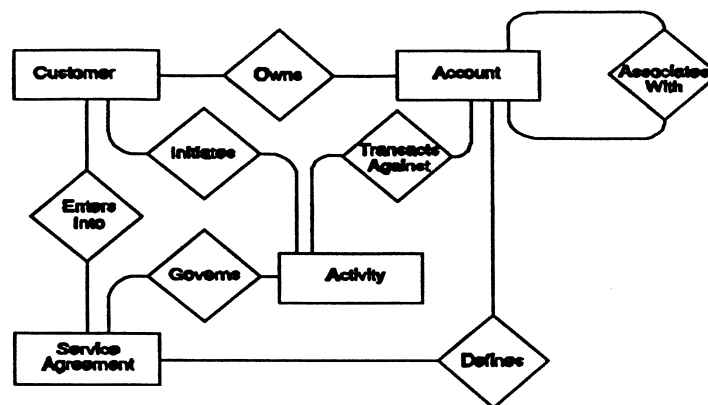


Fig. 3. Global ERD for the retail business of a bank.

⁴ In a top down decomposition of the system, the intent is to select common processes at each level of decomposition by generalization abstractions. The decomposition stops at the lowest functional distinction; that is, at the point at which the functional activities cannot be subdivided into a lower set of distinctive functions.

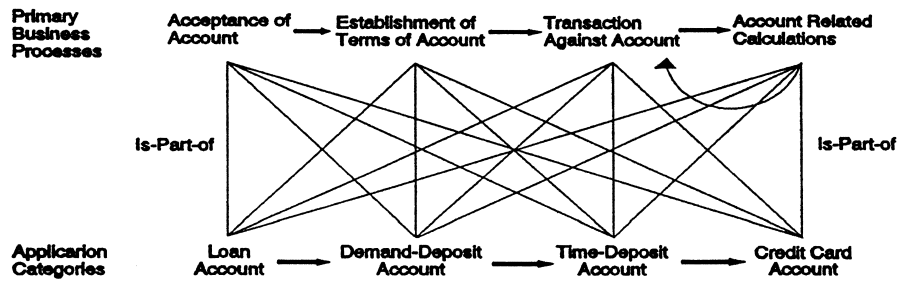


Fig. 4. Primary business processor and application categories.

of the identified processes and its execution cycle is specified. As shown in Fig. 4, the primary business processes common to the global ERD are acceptance of account, establishment of terms of account between bank and customer, transaction against account, and account related calculations. For each of the primary processes, aggregation abstraction is used to identify the functional activities (application categories) of the bank. As shown in Fig. 4, four major accounts representing each of the primary business processing are identified.

Using aggregation abstractions, the application categories are expanded to their subactivities⁵. The subactivities that are repeated in each of the application categories are identified next. Using generalization abstractions, these common subactivities are consolidated to generate task categories, e.g., payment, credit. Then using aggregation abstractions, each of the task categories are decomposed to detail subtask level in order to further identify the common processes. Fig. 5 illustrates the relationships among application categories, task categories, and common processes of the retail subsystem of the financial system shown in Fig. 2.

1.5. Identifying application assets

Application assets deal with logic structures that are repeated many times across applications. By identifying and standardizing these abstractions (logic structures, design information) and defining them logically in a single place, the potential for redundancy in design is reduced. The common processes found in each task category can serve as the basis for reusable design information. By identifying commonality at a high level (across applications), multiple development and maintenance efforts are avoided (within applications). By describing common processes as far up the *is-a* hierarchy as possible, definition/operations are shared with the greatest number of subtypes. By subtyping, commonality between types is accounted for through the supertype, since a subtype is a specialization of its supertype. By standardizing this information in a library, it can be reused across applications. This will avoid the cost associated with

multiple implementation and maintenance efforts. Fig. 6 shows how the common processes can draw on functions of the supertype.

1.6. Identifying data assets

Data assets are associated with application assets. Capturing the meaning of the application assets makes it possible to reuse and maintain such vocabularies. Data assets are defined as semantic knowledge connected with application and task categories. These are the constraints that are placed on objects, operations, and relationships. By capturing this knowledge and standardizing it, the meanings of the applications are captured. Fig. 7 illustrates a typical data asset for the credit loan account in Fig. 5.

2. Frame-based reasoning

A frame provides a structured representation of an object or class of objects. Frames can be used for recognition, interpretation, action, and the prediction of entities or situations on the basis of information obtained from the environment [15]. The use of frames to represent domain entities was explored for design of expert system builder in ESPRIT 96 project [16]. The study reports that by using frames, knowledge modularization and reusability is greatly enhanced in the system. Modularization mechanisms let you manage the complexity of large systems both during their development and maintenance phases. “Frames can be organized into taxonomies using two constructs that represent relationships between frames: *member links*, representing class membership, and *subclass links*, representing class containment or specialization” [17]. A frame can be implemented as a collection of slots⁶. Slots are either *own* or *member slots*⁷. *Situational frames* contain just the rules or

⁵ For each application category, unique and common subactivities can be identified using aggregation techniques. By focusing on identifying all activities (or processes) that are fundamentally different; the primary processes can be identified and defined. This in turn will lead to the identification of the processing cycle.

⁶ Slots are frame components used to store variables or sets of attribute descriptions that characterize the frame. The links allow for two interpretations of the meaning of an *is-a* link, as in ‘A hotel bed *is-a* bed’ and ‘A mattress *is-a* cushion’.

⁷ *OwnSlots* can occur in any frame and are used to describe attributes of an object or class. The value of the slot is not inherited. The *MemberSlots* occur in frames that represent classes and are used to describe attributes of each member of the class, rather than the class itself. The value of the slot is inherited by those of the subclass or is shared by the members within a class [18].

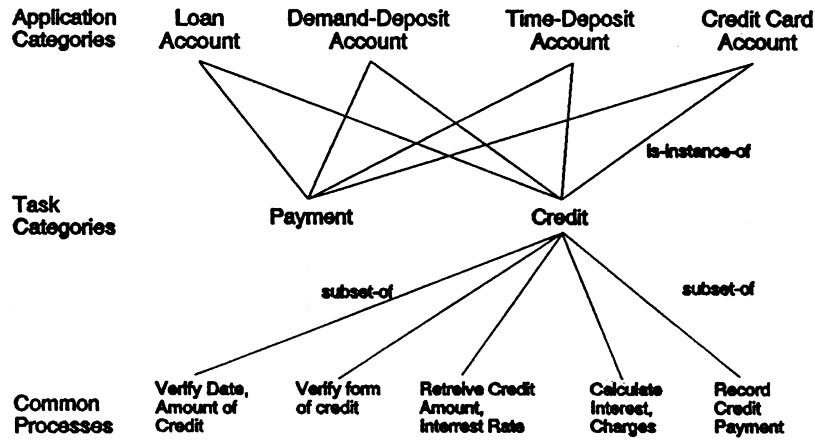


Fig. 5. Relationships among application categories, task categories, and common processes.

facts applicable to a problem and provide some mechanism for relating frames. The primary mechanism used is inheritance through which a subframe can draw upon information from a superframe. *Action frames* consist of slots that specify tasks to be performed. Action frames like situational frames, may also be hierarchically organized through the use of task slots, which may point to other action frames.

Inheritance provides a mechanism by which frames relate to each other. Inheritance is implemented through various links such as *isa*, *ako* (a kind of), and *superc* (super Class). It

is used to establish default values and exceptions. *Value class* and *cardinality reasoning* are constraints on the legal values of a slot. The value class is used to perform membership test. This test is to find out whether a given item is in a class represented by a frame by sending a message to that frame⁸. The frame’s response would determine whether the item is a member or not. Thus, each class in the knowledge base can have its own membership test [20].

Previous research has shown that these features of frames

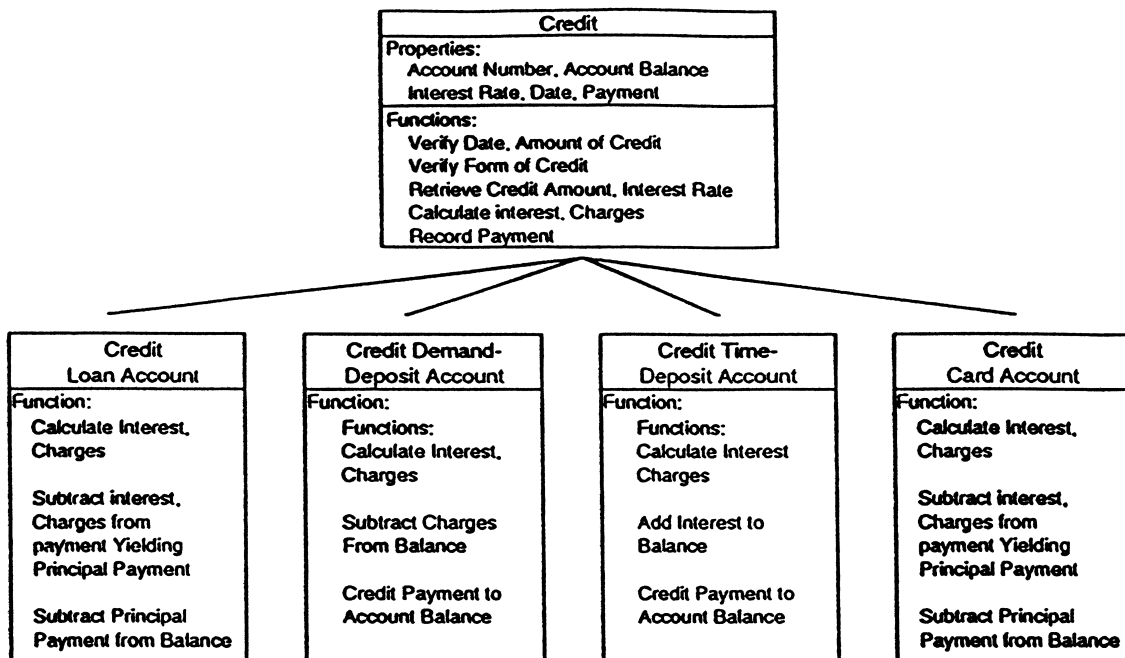


Fig. 6. A graphical representation of an object oriented development for the credit task category.

⁸ Value-class constraint checking procedures understand the semantics of basic set theory operators and numerical intervals. The expert system in [19] uses value class and cardinality to provide constraint checking. This checking is used to determine whether a slot’s value-class and cardinality specifications exclude a given item from being a value in a slot.

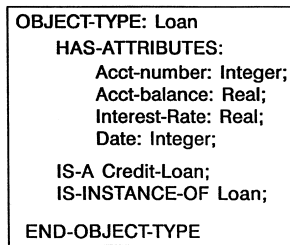


Fig. 7. Data assets.

were useful for constructing, organizing, and controlling the behavior of expert systems using production rules. In addition, by representing production-rules in frames, rules can be grouped into classes. The description of the rules can include attributes of the rules⁹. As a result, frames allows production rules to be organized as easily managed modules (concept bases). These modules contain: (1) the descriptions of the domain entities; (2) structural and behavioural properties of all the entities in a given application domain; and (3) are organized into an inheritance hierarchy where knowledge in one frame is also accessible by all its children. By defining different concept bases, the expert system can be used in multiple applications [9].

3. Implementation in frame-based systems

In this section we first describe how frames are used in an asset-based system, provide some detail on implementation of the system, and finally compare our approach with some of the similar existing methods.

3.1. Using frames in asset-based systems

There are eight major steps in creating data and application assets by using frames. The major task in this system is to identify, create, and store the asset objects in the form of frames in asset library. This task is carried out in the following steps and product of each step is represented by a symbol:

1. Define asset objects as frames, represented by **F**
2. Define class hierarchy for assets/frames by relating them in the form of member links, represented by set **M**.
3. Define class containment in the form of subclass link, set **S**. Class hierarchy is created upon the completion of this step.
4. Define attributes for each frame in the form of *ownlist*, set **OL**, and *memberlist*, set **MI**.
5. Define (specify) class methods (tasks, set **T**) for asset class **F**.

⁹ A rule includes an EXTERNAL.FORM slot containing the rule and a PARSE method for converting the rule into an internal form. The internal form contains lists of expressions. These expressions are the values of the slots CONDITIONS, CONCLUSIONS, and ACTIONS [21].

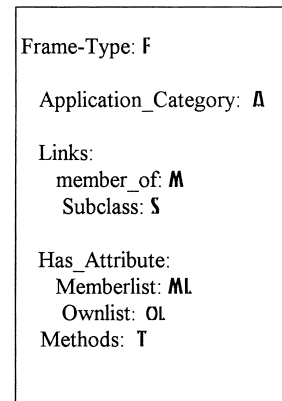


Fig. 8. Structure of a typical frame.

6. Define application categories, set **A**, for each class of object.
7. Use procedures to drive inheritance and access methods among super class and sub class of each new assets/frames and their subclasses.
8. Include asset details (in the form of frame) and details of defined methods for each frame in asset library.

At the end of this process frames contain information and structure as depicted in Fig. 8. The following pages provide more detailed descriptions of the above procedure.

In using frames in the asset-based systems development strategy, objects are represented as frames. Frames are related by two types of links: subclass and member links. The links in the subclass links represent class containment or specialization. However, the links in the member links represent class membership. In the bank example in Fig. 2, Decision Support, Operational, Administrative are examples of objects. Objects are classified in class, subclass, and superclass categories. Within each class of objects there are activities (tasks) and values (application categories) that define the class. The slots in these frames are used for representing tasks and application categories. In Table 1, the superclass (Financial System) is linked with its subclasses (Decision support, Operational and Administrative) via subclass links.

Table 2 represents an example in using member links. Task category (Credit) is linked to its own members (Loan Account, Demand Deposit Account, Time-Deposit Account, and Credit Card Account application category) via member links.

As shown below, the frame *Credit* has common properties such as account number, account balance, interest rate, date, and payment that are shared among the other member frames. Additionally, functions are grouped under the *Credit* frame and are also shared by the member frames. These functions could consist of verifying date, amount of credit, verify form of credit, etc. The common properties mentioned above would be the *Memberslots* for the frame *Credit*.

FRAME: Credit	
Superclass: Retail Member classes: Credit Loan Account, Credit Demand-Deposit Account, Credit Time-Deposit Account, Credit Credit-Card Account	
SLOT	ATTRIBUTES
MemberSlot: ATTRIBUTES	Account-number: Integer Account-balance: Real Interest-rate: Real Date: Integer Payment-amount: Real
MemberSlot: FUNCTIONS	Verify Date, Amount of Credit, Verify Form of Credit, Retrieve Credit Amount, Interest Rate, Calculate Interest, Calculate Charges, Record Payment

Within each frame are two types of slots which describe the attributes of the frame: *Ownslots* and *MemberSlots*. An example of Ownslots is in the frame *Credit Loan Account*. This frame would contain attributes specific to its functions and would draw upon the properties of both *Credit* and *Retail*. Its OwnSlots could calculate payment and update account balance. Both application and data assets can effect attributes and behavior from higher level objects in the hierarchy.

3.2. Reusing assets from asset library

To reuse assets from the asset library for a related application the following steps should be taken:

1. Define the class-type, **F**, application category, **A**, and methods **T**, (or method description) needed for the new application.

FRAME: Credit Loan Account	
Superclass: Retail Member of: Credit	
SLOT	ATTRIBUTES
OwnSlot: HAS-ATTRIBUTES	Principal-payment: Real
MemberSlot: FUNCTIONS	Get Interest, Get Charges, Calculate Principal Payment, Update Account Balance

Situational and action frames are used to define the application and data assets. Within all the application and data assets, value class and cardinality reasoning can be used to place constraints on the legal values with a slot. Therefore, there will be no further need for incorporating error checking within each module. Data Assets are represented by frames using member links. As an example of the credit subclass, the slots within the data assets would be primarily of *MemberSlots*. The specifications for application and data assets for the Credit Task category are shown below;

2. match the requirement specifications (**F**, **A**, and **T**) with the contents of the library to find a possible matching frame.
3. If a match was found traverse the class hierarchies within the frame and check if a complete match is found or modification is needed.
4. If more than one frame from the library is needed to represent the new application a new frame must be created to include the existing frames as its members [22].

While a frame-based implementation approach offers

FRAME: Credit Demand-Deposit Account	
Superclass: Retail Member of: Credit	
SLOT	ATTRIBUTES
OwnSlot: FUNCTIONS	Get Charges, Subtract Charges from Balance, Credit Payment to Account Balance

FRAME: Credit Time-Deposit Account	
Superclass: Retail Member of: Credit	
SLOT	ATTRIBUTES
MemberSlot: FUNCTIONS	Get Interest
OwnSlot: FUNCTIONS	Add Interest to Balance, Credit Payment to Account Balance

FRAME: Credit Credit-Card Account	
Superclass: Retail Member of: Credit	
SLOT	ATTRIBUTES
OwnSlot: HAS-ATTRIBUTES	Principal-payment: Real
MemberSlot: FUNCTIONS	Get Interest, Get Charges, Calculate Principal Payment, Update Account Balance

many advantages, there are some drawbacks that need to be considered. First, frames are pre-structured semantic nets and are not very flexible for representing knowledge in highly dynamic domains. Thus, the structure of a frame and its placement in the overall hierarchy has to be carefully thought out. Besides, the degree to which the domain is defined can influence how inclusive the description of the world being modeled. For example, if a ‘car’ was the smallest atom in a representation, then the system cannot reason about the components that make up a car. Thus, all the relevant information should be included to properly describe a domain. In addition, a frame-based system may draw erroneous conclusions in cases where a given representation has multiple meanings. A frame-based system must be able to handle side effects caused by actions as the program reasons. To prevent unwanted side effects, the domain mapping must be exhaustive enough to handle all possible outcomes [15].

Further, the computational efficiency of a frame-based system can be an important limitation since systems based on unconstrained logic pay a high price in efficiency [23]. The trade-off between the expressiveness of the domain and the efficiency of the logic used in the system’s reasoning need also to be considered. One way for improving the computational efficiency of the system is to define common processes as far up the *is-a* hierarchy as possible. This will allow sharing definition/operations with the greatest number of subtypes. By subtyping, commonality between types is accounted for through the supertype.

Finally, in the case that implementation language permits unrestrained overwriting of the inherited properties care should be taken to avoid debugging problems. These problems may result from frequent changes to functions of a module so that it no longer performs in a manner consistent with its initial definition.

Table 1
Subclass links

Superclass Frame	Subclass Frame
None	Financial systems
Financial system	Decision support
Financial system	Operational
Financial system	Administrative
Operational	Commercial
Operational	Retail
Retail	Credit
None	Utility

3.3. A prototype implementation of frame-based system

Prototypes of this approach have been implemented in XLISP [22]. The XLISP version follows object-oriented concepts of data encapsulation, information hiding, and inheritance.

For implementation, specific consideration may be necessary depending on the language of choice. In implementing the retail subsystem in XLISP (version 2.1). in the above example, the difficulty was how much to decompose each function [24]. In the asset-based systems development strategy, decomposition is used as a method for identifying common processes. To take advantage of inheritance in this implementation, at each level of the decomposition, the common processes were placed in the supertypes rather than subtypes¹⁰. Depending on the language of choice the data and application assets may or may not need to be combined into the same modules. As objects/classes are created and then combined into a program, it is clear that having data assets apart from application assets leads to more work¹¹.

One of the key principles of frame-based implementation is encapsulation function in frames. This works fine if all the information needed by the function is within the module (frame) or can be calculated by the module. It does not take into account possible dependencies that are caused by the passing of data between frames. One way to overcome this is to use global variables, but this would defeat the goal of making the modules ‘black boxes’. Thus, the issue becomes one of scope of variables versus data coupling. In XLISP, variables are considered local in scope within a

Table 2
Member links

Member of Frame	Member Frame
None	Credit
Credit	Demand-deposit accounts
Credit	Loan accounts
Credit	Time-deposit accounts
Credit	Credit-card accounts

¹⁰ Each type (e.g., credit loan) has a supertype (i.e., credit) from which it inherits operations (method, function, e.g., verify amount and payment) and attributes (e.g., implementation for instance variables such as payment-amount).

¹¹ It takes more work to include the two separately than it does if they are one entity.

class. That means data assets are always being locally defined within the scope of the class. By making the data assets static, either parameters need to be passed (which means more attention will have to be paid to data coupling), or some encapsulation properties need to be given up.

3.4. Organizational requirements to use asset-based systems

The asset-based systems development strategy and the proposed method require a corporate infrastructure that encourages and rewards software reuse. It requires that top management understands the critical role of software reuse; it requires participation of project management and software experts in strategic information systems planning. These are essential requirements if top management is to address non-technical issues (e.g., legal and proprietary rights, compensation for the developers of reusable parts, internal cost apportionment methods for purchasing reusable parts, etc.) associated with software reuse. These are equally as important as the technical issues.

3.5. Comparison of frame-based implementation and other approaches

In DRACO [25], domain analysis is used to identify sets of common concepts in a particular application domain. In this approach, common concepts are recorded using a domain-specific language. Neighbors suggest that each domain yields a different domain representation language. The common concepts are given to a domain designer who specifies implementation for these concepts already identified in that domain. System requirements for a new application are considered in light of concepts already identified. By developing several systems for the same domain, a reuse of the domain analysis is achieved. Neighbors did not propose a theory or technique to perform domain analysis. He did, however, distinguish between two activities: domain analysis and domain design. The design stage involves casting the results of the domain analysis in a form that can be usable by the DRACO technology: a domain language parser, a domain language prettyprinter, source-to-source transformation for the domain, and components for the domain must be specified to create a usable DRACO domain. However, as detailed by Horowitz and Munson [21], this approach is inherently limited by a potential growth in domain languages, each of which must be mastered simultaneously

Arango [2] and Fisher [17] also emphasize the importance of domain analysis and domain-related abstractions in making reuse and redesign possible. They suggest the need for an intelligent design environment that keeps track of the levels of abstractions in a software design process. Similar to the asset-based development strategy, Fisher suggests that the levels of abstractions need to be generated throughout the entire process, from determination of system level requirements until implementation at the language level.

Such an intelligent system facilitates reuse and redesign by allowing the user to change intermediate abstractions to form a slightly different system. But to do this the design of the intermediate abstraction levels must be an integral part of the software design process [17]. However, Fisher offers no specific strategy for: (1) identifying the common data and processes within across applications; and (2) implementation of the intelligent design environment.

The differences between an asset-based approach and domain analysis follows. Arango [2] sees reuse as a learning system. In his proposed model, software development is a self-improving process that draws from a knowledge source which is named reuse infrastructure and is integrated with the software development process. Reuse infrastructure consists of reusable resources and their description [2].

In the reuse environment promoted by Arango, using reuse infrastructure and by specification of the software to be built, implementation of desired software can be achieved. The software produced is compared to the input of the system (specification of the system). The lack of a formal and systematic methodology is the primary issue in domain analysis [2].

Basset has also proposed a frame-based approach to software reuse. His approach attempts to provide a foundation for construction of software by formalizing frames using a general frame syntax [26].

Besides the frame-based implementation, a functional approach can be used. In the functional approach, the generalized application assets are defined by specifying input, output, and detail-processing steps in the form of a code skeleton. Then the related application programs are constructed by specifying those parts that have been left open in the generic application assets. For classification purposes, generalization/specialization rules can be specified for each task category. These specialization rules can consist of rules for constants, procedures, and interfaces, but each task category requires specification of different properties by a specialization rule. In the approach by Mitermeir and Oppitz [27] information about each task category is reused by completing it with different specialization rules for each task or application instance.

4. Summary and conclusions

This paper has shown how a frame-based representation system can contribute to the asset-based systems development strategy. Previous work on using frames has been used to identify the properties of frames. Although frame-based implementations are not without drawbacks, this paper has shown that frames lend themselves very well to an asset-based systems development strategy. Frames allow for the integration of data and process modeling advocated in the asset-based strategy. By this integration, data and process redundancy can be reduced. This will result in the elimination of multiple development and maintenance efforts.

Although the asset-based systems development strategy is effective in identifying application and data assets, care must be taken in the implementation stage to take advantage of especial features of the language selected. In building a frame-based representation, for example, careful decisions were made on when to allow frame slots to be inherited as MemberSlots and when to use them as OwnSlots with no inheritance. Further, this paper has shown that frames can contribute to the system's reasoning activities, and can organize and direct those activities. By using frames to integrate data and process, data and process redundancy is reduced. This results in the elimination of multiple development and maintenance efforts. Frames also permit the commonalities among data and processes to be shared. Future research needs to address the cataloguing and library structure to support a reuse system based on the frame-based implementation.

References

- [1] M. Zand, M. Samadzadeh, Software Reuse – Issues and trends, invited editorial, *Journal of Systems and Software*, September 1995.
- [2] G. Arango, Domain analysis methods, *Software Reusability*, eds. W. Schaeffer, R. Prieto-Diaz, M. Matsumoto, Ellis Horwood, New York, 1993, pp. 17–49.
- [3] G. Arango, M. Griss, W. Tracz, M. Zand, Software reuse: issues and perspectives, *Proceedings of ACM-SAC'94*, Software reusability track, Phoenix, AZ, March 1994, pp. 596–598.
- [4] T. Biggerstaff, C. Richer, Reusability framework, assessment and directions, *IEEE Software* 4 (2) (1987) 41–48.
- [5] I. Jacobson, M. Griss, P. Jonsson, *Software reuse*, ACM Press and Addison Wesley, New York, 1997.
- [6] J.S. Pouling, Populating Software Repositories: Incentives and domain-specific software, *Journal of Systems and Software* 30 (3) (1995) 187–199.
- [7] E.A. Karlson, *Software Reuse: A holistic approach*, Wiley Series in Software Based Systems, 1995.
- [8] T. Biggerstaff, A. Perlis, Forward: Special issue on software reusability, *IEEE Transactions on Software Engineering* 10 (5) (1984) 474–476.
- [9] J. Karimi, An asset-based systems development approach to software reusability, *MIS Quarterly* 14 (2) (1990) 179–198.
- [10] U. Apte, C.S. Sanakar, M. Thakur, J.E. Turner, Reusability-based strategy for development of information systems: Implementation experience of a bank, *MIS Quarterly*, December 1990, pp. 421–433.
- [11] T. Biggerstaff, M. Lubars, Recovering and reusing software designs, *American Programmer*, March 1991, pp. 2–11.
- [12] R. Hull, R. King, Semantic data base modeling: survey, applications, and research issues, *ACM Computing Surveys* 19 (3) (1987) 201–260.
- [13] J.M. Smith, D.C. Smith, Data base abstractions: aggregation and generalization, *ACM Transactions on Data Base Systems*, June 1997, pp. 105–133.
- [14] P.P. Chen, The entity–relationship model: toward a unified view of data, *ACM Transactions on Database Systems* 1 (1) (1976) 9–36.
- [15] R.J. Schalkoff, *Artificial intelligence: an engineering approach*, New York: McGraw-Hill Publishing Company, 1990.
- [16] M. Cherubini, F.-L. Alessandra, P. Torrigiani, M. Zallocco, An integrated expert-system builder, *IEEE Software* November 1989, pp. 44–52.
- [17] G. Fisher, Cognitive view of reuse and redesign, *IEEE Software* 4 (4) (1987) 60–72.
- [18] G. Caldiera, V. Basili, Identifying and qualifying reusable software components, *IEEE Computer*, 61–70, Feb. (1991).
- [19] R. Fikes, T. Kehler, The role of frame-based representation in reasoning, *Communications of the ACM* 28 (9) (1985) 904–920.
- [20] R.L. Joos, An Intelligent environment for the use and design of reusable modules, *Expert Systems for Information Management* 3 (1) (1990) 25–47.
- [21] E. Horowitz, J.B. Munson, An expensive view of reusable software, *IEEE Transactions on Software Engineering* 10 (3) (1984) 477–487.
- [22] M. Zand, J. Titus, Implementation of asset-based reuse system: a frame-based approach, Technical Report, Math. and Computer Science Dept., UNO, 1994.
- [23] F.F. Luger, W.A. Stubblefield, *Artificial intelligence and the design of expert systems*. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1989.
- [24] W. Schaeffer, R. Prieto-Diaz, R. Matsumoto, *Software reusability*, Ellis Horwood, New York, 1993.
- [25] J.M. Neighbors, The DRACO approach to constructing software from reusable components, *IEEE Transactions on Software Engineering* 10 (5) (1984) 567–574.
- [26] P.G. Basset, Frame-based software engineering, *IEEE Software* 4 (4) (1987) 9–15.
- [27] R.T. Mittermeir, M. Oppitz, Software bases for flexible composition of application systems, *IEEE Transactions on Software Engineering* 13 (4) (1987) 440–460.