A Data/Knowledge Paradigm for the Modeling and Design of Operations Support Systems

Vijay K. Vaishnavi, Senior Member, IEEE, Gary C. Buchanan, and William L. Kuechler Jr.

Abstract—This paper develops the Smart Object paradigm and its instantiation, which provide a new conceptualization for the modeling, design, and development of an important but little researched class of information systems, Operations Support Systems (OSS). OSS is our term for systems which provide interactive support for the management of large, complex operations environments, such as manufacturing plants, military operations, and large power generation facilities. The most salient feature of OSS is their dynamic nature. The number and kind of elements composing the system as well as the mode of control of those elements change frequently in response to the environment. The abstraction of control and the ease with which complex dynamic control behavior can be modeled and simulated is one of the important aspects of the paradigm. The framework for the Smart Object paradigm is the fusion of object-oriented design models with declarative knowledge representation and active inferencing from AI models. Additional defining concepts from data/knowledge models, semantic data models, active databases, and frame based systems, are added to the synthesis as justified by their contribution to the ability to naturally model OSS at a high level of abstraction. The model assists in declaratively representing domain data/knowledge and its structure, and task or process knowledge, in addition to modeling multilevel control and interobject coordination.

Index Terms—Operations support systems, modeling control abstractions, knowledge bases, databases, object model, semantic data model, production system model.

1 INTRODUCTION

This paper develops the Smart Object paradigm and its instantiation, which provide a new conceptualization for the modeling, design, and development of an important but little researched class of intelligent information systems, Operations Support Systems (OSS). OSS is our term for systems which provide interactive support for the management of large, complex operations environments, such as manufacturing plants, military operations, and large power generation facilities. The most salient feature of OSS is their dynamic nature. The number and kind of elements composing the system as well as the mode of control of those elements change frequently in response to the environment. Control structure and flow in OSS shares many of the characteristics of adaptive real time machine controllers, which require "... the consideration of requirements, goals and events of system characteristics with varying degrees of uncertainty, and which cannot be completely specified a priori" [35]. This presumes a meta control level "... unlike conventional controllers in that the procedure specifying the controller, termed the control law, is not given explicitly but is inferred from the knowledge base at each update time" [35].

The Smart Object conceptualization and its instantiation synthesize elements from multiple paradigms, specifically object orientation and production systems, to create a conceptual model optimized for OSS. In addition to drawing from multiple paradigms, the Smart Object conceptualization adopts functionality from multiple disciplines. Data modeling, knowledge engineering, and general conceptual modeling provide many of the grounding concepts. Software engineering and information systems development frameworks for specific classes of systems. This parallels the recent attention to general software architectures [20] and specifically the architectures for domain-specific adaptive intelligent systems [27], [28].

The contribution of the paper falls into three areas:

1) The result of a general requirements analysis of an important class of information systems, OSS, which is developed as motivation for the Smart Object concept.
2) The Smart Object paradigm, both as a highly synthetic data/knowledge fusion that addresses specific problems in OSS development, and as a set of concepts with broader applicability in the knowledge based systems development area. For example, though we develop a metalevel architecture in Smart Objects primarily to accurately model operations environments, others within the knowledge and data engineering community have proposed metacontrol ar-

1. Recounting his experience with a large, knowledge based system project, Fredrick Hayes-Roth indicated, "We have gathered much evidence that knowledge-based computing cannot stand apart from conventional data processing techniques and concerns" [27].
chitectures for production systems as a solution for severe maintenance problems in large expert systems [32]. The segmented object architecture of the Smart Object paradigm addresses this problem, without introducing the complexity of multiple programming paradigms (declarative for knowledge representation, procedural for control).

3) The concept of control modeling, that is the abstraction of the elements of the system that constitute the control structure of a system and its dynamics, and a technique for expressing models of control in production systems.

A nonrigorous narrative description of the basic structure of the Smart Object paradigm is given in Fig. 1.

- Real world systems are conceptualized as a network of minimally coupled active entities called Smart Objects (SOs).
- The network of SOs fully partitions all knowledge contained in the model of the system.
- Different knowledge representations can be used to optimally support different tasks (procedural vs. declarative knowledge, for example).
- SOs can inherit knowledge and data structures analogously to facilities available in most OOP languages.
- SOs are themselves partitioned into: domain knowledge, state information, control knowledge, and interface information.
- Control is abstracted by explicit control knowledge and an interobject control architecture which defines the operationalization of the control knowledge. The object structure together with inheritance give support to reusable domain frameworks.
- A multimodal, logic engine interprets the modeled knowledge. Continuous inference cycles derive new knowledge and system states from the existing state, and information from the system environment.

Fig. 1. A narrative description of the Smart Object paradigm framework.

The manner in which the semantics of high level designs are actively interpreted by a Smart Object system is diagrammed in Fig. 2. A more detailed description of the paradigm is given in Section 3.

The logical view portion of Fig. 2 depicts a typical object-oriented system model, as would be generated by the systems analysis phase of many object-oriented analysis methodologies. The architectural view portion of Fig. 2 depicts the active interpretation of the same model, after population of the Smart Objects that compose it with declarative control and domain knowledge. The transition from the Smart Object paradigm to an instantiated OSS, that is, from system conception to a functional, fully articulated support system, follows the steps in Fig. 3. The steps constitute an informal methodology for the realization of working OSS from the paradigm, given a new domain for which there has been no prior analysis or development.

Fig. 3 is intended to be read top to bottom, left to right. Beginning with the general paradigm as a manner of conceptualizing a system, use of the paradigm proceeds by the actions in the Transition Mechanism column to make the model progressively more concrete. As the model becomes more concrete, the applicability of the model is given in the Abstraction Level column, and increasing capabilities of the model are summarized in the Functionality column.

The remainder of the paper will be devoted to elaborating the general concepts set out in Figs. 1 and 2 as well as their utility. The structure of the paper follows the progression of Fig. 3, that is, from the general specification of the paradigm through its instantiation as a metamodel, instantiation of a domain model based on the metamodel, and finally, a concrete example taken from an OSS model for a nuclear power facility.

Though Smart Objects will be shown to be applicable to the modeling and development of other types of systems, such as large expert systems, the primary motivation for the paradigm is the unique character of complex operations environments, which is developed in the next section (Section 2). The Smart Object paradigm is broadly defined in Section 3, followed by a direct comparison of the paradigm with a number of earlier data/knowledge fusions. In Section 4, the paradigm is instantiated to a metamodel, the Smart Object Language (SOL), and the instantiation continues in Section 5, where a control architecture for a domain is developed in SOL to yield a general system framework for the domain. A section of a model of a nuclear power plant is used in Section 6 to show components of a working Operations Support System in SOL and to elaborate on the features of SOL in the context of the example. The concluding section discusses current shortcomings of the paradigm and directions for further development.

2 PROBLEM DEVELOPMENT: THE CHARACTER OF COMPLEX OPERATIONS ENVIRONMENTS

Many complex environments, such as power plants, paper and chemical manufacturing plants, military operations, and the space program, consist of a large number of operations which interact to accomplish system objectives. These environments are governed by knowledge in the form of records, formal procedures, informal procedures and established rules for governing the interaction of operations. Much of the data is constantly updated by the vast number of functional activities that accomplish the operational objectives. The sheer volume of knowledge and extensive interaction between operations places a heavy cognitive burden on personnel who need to make operational decisions based upon the current global status of the system. No tra-
ditional class of information system (expert systems, executive information systems, etc.) addresses the depth of interactive, global support required [56], so we have coined the term Operations Support Systems (OSS) for a new class of systems intended to perform that function.

The focus of OSS is to support the routine functioning of systems operations at several levels in the organizations simultaneously. Maintenance and service functions for a large airline provide an example from a commercial enterprise. The components of such a complex system include materials, service managers, workers, equipment, company procedures, and externally imposed (FAA) regulations and procedures. At the managerial level, the service manager must coordinate activities to ensure timely and correct services. At the operations level, maintenance personnel must perform their duties in accordance with complex procedures. At the operations level, maintenance personnel must perform their duties in accordance with complex procedures. Both organizational levels would benefit from the global perspective supplied by OSS [56]. The need is so great for integrated, global support that defacto OSS have been and continue to be developed without any formal methodology, frequently as multiple, separate expert and data processing systems, loosely coupled through the manual efforts of midlevel managers [57]. A prolog to a formal development methodology for OSS is an analysis of the unique characteristics of such systems, and a functionally complete domain model.

Smart Objects originated as a solution to the problems encountered while attempting to model a complex operations environment (a nuclear power plant) with PROLOG [6]. Our intention was the implementation of an active, intelligent support system for this environment. Subsequent efforts showed that the approach was correct but that a more focused tool was needed than a general logic language [6], [44], [58]. Gradually, a set of characteristics emerged that we believe are both necessary, that is, are minimally required to model knowledge of the structure and behavior of large, complex OSS, and implementationally practicable, that is, they provide a model that can be used by business application programmer/analysts for the development of effective, real-world systems. OSS are iteratively developed from initially fuzzy specifications, and must be easily maintained so that they can evolve in parallel with the activities they support. As our experience with PROLOG showed, a model (first order logic) may be mathematically complete, but is unusable for routine systems development due to the cognitive limitations of human designers.

2.1 Attributes of a Conceptual Model for Operations Support Systems

During the systems analysis phase of the power plant modeling project it was determined that OSS have the scale problems of large management information systems coupled with the domain knowledge and advisory requirements of decision support systems (DSS) and expert systems (ES). Though incorporating a substantial knowledge and inference component, and similar in some functions to expert systems, information systems capable of supporting operations planning and control must have a far greater scope and power than expert systems as currently conceived [24].

Fig. 4 summarizes the attributes we feel are essential to any conceptual model of support systems for complex operations environments. Each of the major points in the figure is discussed in turn.
Conceptual Attributes
- knowledge associated with operations (tasks)
  - data and processes
  - constraints on data or processes
  - prerequisites and temporal relationships
  - inferential relationships
- adaptive inferencing
  - support for multiple inference strategies
  - defeasible logics to support nonmonotonic reasoning
- structural relations between operations
  - metacontrol capability for dynamic alteration of structure and behavior
- control modeling
  - monitoring the status of operations (tasks)
  - responding to changes in system activities
  - modeling system behavior
- management of conceptual complexity
  - modeling elements and relations can be recursively applied for hierarchical concept building

Functional Attributes (reflect the intended utility of the model)
- management of development complexity through software engineering principles
- executable nature
- easily modifiable, encouraging prototyping, reuse

Fig. 4. Attributes of a conceptual model for OSS.

First, the model must be able to capture explicitly the various types of knowledge associated with the complex tasks that affect decision making. This knowledge includes data and the operations on data that are a part of a task as well as constraints that apply to tasks and data. These constraints might be in the form of prerequisites or temporal restrictions on when or in what sequence the operations that make up the task must be performed. In addition, there may be inferential relationships derived from established rules and regulations that must be represented in the system.

The systems that OSS model are goal driven, highly adaptive composites of totally manual procedures, and procedures driven by partial information from existing automated systems. An attribute of complex operations environments that is frequently encountered and difficult to model is their dynamic reorganization in response to perturbations in normal operation (exception conditions). The handling of exceptions frequently results in the re-structuring of control flows within the hierarchy, and may result in the creation of temporary structural entities which endure for the duration of the exception, and are then replaced by the old structure and control flows on resumption of normal operations. Work on adaptive systems in both the artificial intelligence community, and the software engineering community specifies “reflectivity,” embodied in an architectural metalevel, as a requirement for adaptivity [35], [45]. A reflective system is one that explicitly represents its own operations and structures, and interprets these in the performance of its processes [39]. Though lacking the time constraints of real-time machine control systems (OSS interact with humans, rather than directly controlling other systems), the concept of inferring control disposition in response to changes in the environment is common to OSS and adaptive control systems [35]. The adaptive structural reorganization required by OSS is isomorphic to the concept of a dynamic interconnection topology discussed with respect to ACTOR models of concurrent computation [1]. It is also essential that a model for OSS be capable of representing this behavior.

Complex operations environments are invariably hierarchical; the hierarchy serves to structure information flow (control flow), and to partition functional complexity and decouple the large number of functional tasks occurring at different levels of the hierarchy. Functional status information flows up the hierarchy and is aggregated into progressively more general and encompassing system states. A paradigm intended to model support systems for these environments should naturally and effectively model hierarchical structure and the control aggregation implications of hierarchy in the target environment.

Since an operations system is, in essence, a web of interacting tasks, the model must be able to explicitly represent not only the knowledge associated with an operation but also the structural relationships between operations. In an operations support system (OSS) environment, it is extremely difficult for personnel to know the status and condition of all system activities that affect the current decision-making situation. By capturing knowledge associated with an operation and the critical relationships with other tasks that affect a specific operation, an OSS can make personnel aware of these activities and how these activities may change based on the status of other system events.

Thus, in addition to the knowledge structures appropriate for OSS, the model should embody the control abstractions common to the domain of OSS. These control abstractions contain knowledge about how to make use of the conceptual structures representing operations. This knowledge includes how to monitor and track the status of critical relationships and what to do if certain constraints are not met. Having these control abstractions built into the conceptual model will allow these control abstractions to be carried across OSS applications; domain frameworks offering very high levels of reuse are made possible. In addition, the model must allow the incorporation of customized control knowledge representing the behavior of a particular OSS at different abstraction levels.

In order to effectively and efficiently develop operations support systems at a high level of abstraction, there is a need for both a conceptual data/knowledge model and a language that supports the abstractions of the model. The need to develop a model at a high level of abstraction is especially acute for OSS’ since specification of a stable set of requirements for such systems cannot be done in advance of system implementation. The only feasible and effective way of developing such systems is through prototyping/simulation and incremental development of the complete system at a level of abstraction high enough to allow end-users to participate in the development process.

In order for the model to be an effective prototyping and simulation vehicle, it must have an executable nature so that users and developers can evaluate the simulated system. Many systems which capture expert knowledge can only be validated through prototyping [41]. Finally, to effectively manage the complexity associated with system development, a model for OSS should embody and enforce the software engineering principles [3], [4], [52] of abstraction, modularity, information hiding, and localization.
A comparison of OSS attributes to existing data, knowledge, and design model characteristics has been assembled as Fig. 5, below (cf. Figs. 4, 8, and 12), and indicates why we were unable to adopt the models to the task of OSS development without major changes.

While the rankings of the various models, like all rankings, has a subjective element, we have tried to adhere to published, acknowledged assessments of the models. For example, data modeling is the raison d’être of semantic data models, and data abstraction is an acknowledged strong point of OO design models; production systems and first order logic are typically regarded as inferior to both those models in this capability. Likewise, semantic data models were not designed to specify complex sequential processes, whereas object-oriented design has precisely that intent; large systems built with logic based or production system languages such as OPS5 are notoriously difficult to maintain [32], [27], [57].

With respect to control modeling, an aspect critical to OSS, none of the models or paradigms provides control abstractions which specifically addresses the representation of system behavior at different abstraction levels. OSS behavior under exception conditions is sufficiently dynamic to require well developed metalevel control for accurate representation. These weaknesses in control representation and abstraction can be overcome by incorporating concepts from a knowledge representation model (e.g., the production system model) into a paradigm that incorporates concepts from a variety of existing, familiar models. Within the framework outlined above, the paradigm incorporates concepts from older, more widely used models provides partial validation of them. The paradigm supports the instantiation of executable models that are capable of abstracting dynamic control situations.

In the next section of the paper, a basic framework for the Smart Object paradigm is presented followed by a discussion of the concepts from existing data/knowledge paradigms that are incorporated into the paradigm using the basic framework. The paradigm is then compared with other existing related paradigms.

### 3 Paradigm Definition and Literature Review

#### 3.1 Paradigm Framework

The Smart Object paradigm, like all models, is a conceptualization: a set of concepts and terms for defining some subset of the real world: the universe of discourse. Formally, a conceptualization is a set of elements, functions, and relations that are used to define the things in the universe of discourse and the way in which they interact [22]. Fig. 6 presents a more detailed description of the Smart Object conceptualization framework. The framework is motivated by the need to purposefully bring together concepts from knowledge-representation, semantic, hyper-semantic, and active database models.

Note that by virtue of having the control behavior derived from the interpretation of knowledge elements by the logic engine, control has been abstracted to the same level as domain inferences; that is, control becomes “just another problem to be solved” [45]. This concept is also a key element of the SOAR model of generally intelligent behavior [54], and indicates the generality of the paradigm.

#### 3.2 Existing Model Concepts Fused into the Smart Object Paradigm

Within the framework outlined above, the paradigm incorporates concepts from a variety of existing, familiar models and paradigms, as summarized in Fig. 7; the use of these concepts in older, more widely used models provides partial validation of them. The paradigm supports the instantiation of executable models that are capable of abstracting dynamic control situations.

Object-oriented design models are the strongest influence on the Smart Object paradigm. System modeling with object based program simulations is the origin of the Object paradigm and is the core of Smart Objects. The concept of a metalevel architecture, allowing dynamic modification of the interpretation of program code, was first defined in an object model in Smalltalk [13]. The close conceptual and implementational linkage allows many of the system analysis and design methodologies developed originally for OO [11], [4] to be directly applicable to Smart Object systems. Likewise, principles for formal proofs of correct system decomposition [47] are directly applicable to systems composed of Smart Objects.
A network of smart objects, $S$, encapsulate knowledge of the domain of discourse. The objects have an internal structure. A Domain Methods section, $M$, which contains the set of domain (application) knowledge elements.

An Attributes section, $A$, which contains the set of variables representing observable states of the object.

A Monitor section, $C$, consisting of the set of control knowledge elements and control states which abstract the object specific (application level) control and coordination behavior of each element of $S$.

An Interface section which contains a set $I$ of communication and coordination knowledge elements for each object. The application semantics of the logical relationships between Smart Objects in the system is abstracted in this section.

A control architecture $H$ consisting of data structures and knowledge which abstract the general control behavior of the elements of $S$. Different control architectures can be defined to reflect the different control mechanisms appropriate for different application domains, that is, $H$ is specific to a domain framework.

A logic engine, $L$, which is capable of making inferences from knowledge elements in $S$ and state information, and altering the states of individual Smart Objects, their control flows, and the structure of $S$:

- During any inference cycle, only a portion of the total application knowledge elements, from $M$, will be active (applicable).
- At each cycle, $L$ may infer a new active set.
- During any inference cycle, only a portion of the total control knowledge elements, from $C$ and $H$ will be active (applicable).
- At each cycle $L$ may infer a new active set.

As a result of any inference cycle, $L$ may create or remove a Smart Object from the total system. Newly instantiated objects must be derived from an existing object (parent) and inherit the knowledge elements ($M$, $L$, $C$) of the parent, the relations of the parent (to already extant elements of $S$), and the state variables of the parent (from $A$ and $C$) with all variables initially set to the null state.

Fig. 6. Smart Object paradigm framework.

Semantic data models (SDMs) [30], [48], [43] offer powerful abstractions such as generalization, aggregation, classification, and association to model both data and relationships. The abstractions are used in the definition of attributes and their relationships. Active database models [14], [17], [40], [55], [60] allow designers to specify actions to be taken when certain conditions arise. This is usually done in the models by the attachment of triggers (i.e., rules) to data items.

Modeling of control has received more prominent attention in languages for expert system design such as those based on production rules [5], [26], predicate logic [21], frames [19], and generic tasks [8], [10]. Each such language has constructs for representing knowledge for a given problem domain and a control mechanism (inference engine) for using or manipulating the knowledge structures to produce solutions or answers to questions.

The need for Smart Objects, as representations of tasks, to dynamically change sequence and redirect the flow of results in response to environmental changes is analogous the situation in which multiple, parallel computing elements are dynamically reconfigured in response to computational load changes in the ACTOR model of concurrent computation. The solution detailed in that literature is termed a dynamic interconnection topology [1]. Likewise, the paradigm also borrows from the computer science domain, the concept of metadevelopment environments, or software architectures [20] which are used to develop domain specific frameworks for a class of system solutions [28], [27].

The notion of object-oriented languages that allow their programs to reason about themselves or to reflect on their computations has given rise to the concept of “reflection” and “meta-level architectures” in object-oriented programming [31]. This concept is related to that of “exception handling” in programming [15] and “metarules” in rule-based expert systems which dynamically change the interpretation of a knowledge base based on new facts [23], [32].

In the prior discussion, the concepts which have been incorporated into the paradigm (cf. Fig. 7) and which enable high level modeling of OSS, have been defined in terms of the model in which they originated. The table in Fig. 8 relates those concepts to the Smart Object paradigm framework element in which they are contained in an instantiated Smart Object system.

With reference to Fig. 8, the engineering of the knowledge representation (used to represent the knowledge elements of $M$, $A$, $C$, $I$) and the control architecture, $H$, is a design problem; each element progressively constrains the other as they emerge from the design process. We wish to note that there are potentially an infinite number of instantiations of the paradigm, and many are quite feasible. The idea of using a segmented blackboard architecture [45] for both the smart object knowledge partitioning, and as a communication/control structure was considered elegant and appealing. However, the better researched and theoretically grounded production system model, in conjunction with LIFO stacks of control records for each Smart Object, made a better choice for our first exploration of the paradigm as SOL (described in Sections 4 and 5).

### 3.3 Related Paradigms

Prior to beginning a review of related previous work, it is appropriate to restate that the Smart Object paradigm is not a specific data/knowledge model, but rather an architectural specification for the development of data/knowledge models optimally useful for the task of modeling and developing OSS. A requirement for both the validation of the knowledge components of the system, and for a working artifact, is that the models be executable. Satisfying this requirement entails the use of: 1) heuristics, 2) defeasible and modal logics, and 3) nonmonotonic reasoning, and has the purposeful, but less than ideal consequence of making the systems difficult to prove correct in a rigorous manner. The nature of the compromise between rigor and breadth of utility is discussed extensively in [18] in terms of the constraints of liveness (motion toward the goal) and safety (provable correctness). Though ideally both constraints should be satisfied, for working systems deployed in complex, real-world environments, liveness is the stronger constraint, since a system lacking a formal proof of correctness can still have a high likelihood of utility, whereas one exhibiting not (liveness) is unlikely to be considered successful by its end users.

A number of previously published data/knowledge fusion models/paradigms are quite broad in their capabilities, and exhibit some similarity to SOL, the rule based in-
VAISHNAVI ET AL.: A DATA/KNOWLEDGE PARADIGM FOR THE MODELING AND DESIGN OF OPERATIONS SUPPORT SYSTEMS 281

OSS Attribute

<table>
<thead>
<tr>
<th>Data/Knowledge or Architectural Model</th>
<th>Concepts Incorporated into Smart Object Paradigm</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-oriented design models</td>
<td>object-oriented system conceptualizations, inheritance, encapsulation, situationally overridable methods (polymorphism), design and implementation methodologies, simulation</td>
<td>[4], [11], [47]</td>
</tr>
<tr>
<td>Semantic data models</td>
<td>generalization, aggregation, classification, association; dynamic data modeling, integrity constraints; conceptual language</td>
<td>[30], [43], [48], [49], [50]</td>
</tr>
<tr>
<td>Active databases</td>
<td>data semantics, triggers on activities access</td>
<td>[14], [17], [40], [55], [60]</td>
</tr>
<tr>
<td>Production systems; frames</td>
<td>control abstraction</td>
<td>[5], [19], [21], [26]</td>
</tr>
<tr>
<td>Generic tasks</td>
<td>reusable macro logic/control structures</td>
<td>[8], [10]</td>
</tr>
<tr>
<td>ACTOR</td>
<td>dynamic interconnection topology</td>
<td>[1]</td>
</tr>
<tr>
<td>Generalized software architectures</td>
<td>metalevel development of domain specific frameworks</td>
<td>[20], [27], [28]</td>
</tr>
<tr>
<td>Metalevel control architectures</td>
<td>reflection; dynamic adaptation of control to external events</td>
<td>[15], [23], [31], [32], [35]</td>
</tr>
</tbody>
</table>

Fig. 7. Concepts from existing models incorporated into the Smart Object paradigm framework.

Fig. 8. Relationships between Smart Object paradigm elements and OSS attributes.

stantiation of the Smart Object paradigm, which will be presented in Sections 4 and 5 of the paper. The characteristics of these systems, our reasons for considering them less than ideal for OSS modeling, and references to extended presentations of the models/paradigms are summarized in the table in Fig. 9.

Of the models/paradigms summarized in Fig. 9, we consider Active Knowledge/Data Language (KDL) the most similar to the Smart Object paradigm. Active KDL is an object-oriented database approach to system simulation, extended with scripting, a form of programming. However as Potter, Miller, and Krzysztof [51] point out in their paper, Active KDL is a query driven simulation system, differing in capability and intent from both knowledge- and object-based simulations. An active KDL model is quiescent until an external agent presents the system with a query. Other significant differences from the Smart Object paradigm are the use of separate scripting, and data definition and constraint specification languages, both of which are procedural rather than declarative.

Vesti, Nordbo, and Solvberg [59] directly address the importance of control modeling in the development of intelligent systems. Their context is the COMEX system, which is an object structured knowledge modeling tool which uses production rules as the knowledge representation. COMEX would bear a close resemblance to the SOL Smart Object paradigm instantiation if Methods were the only Smart Object section. Vesti et al. draw the same conclusion with respect to control modeling in production systems that Ishida, Sasi, Nakata, and Fukuhara [32] do: Implementing control directly in production rules is cumbersome, inefficient and confusing, and so an external mechanism is required to implement control. Vesti et al. propose a static, first order logic control, programmed via a graphic notation that portrays logical (and, xor, or) links between object representations. Ishida et al. propose a separate procedural control language to augment declarative knowledge representation. The Smart Object solution to control in production rule based systems is conceptually completely different from that proposed by either Vesti et al. or Ishida et al.: The same language is used for control and domain knowledge expression, but the control modeling rules are separated from the knowledge representation rules not only syntactically, by use of a separate language section, but logically and implementationally by the use of separate inference phases.

4 Instantiating the Paradigm: Smart Object Language (SOL)

In this section, a metamodel, able to efficiently describe systems common to an application domain, OSS, is developed from the general Smart Object paradigm. The knowledge representation mechanism, a production rule based language, and the data structure representing the control architecture, a LIFO stack, are defined. The terse definition given in this section is elaborated on in Section 6, when the language elements are discussed in the context of an example SOL system.


### Table: Origins and Shortcomings for OSS Modeling

<table>
<thead>
<tr>
<th>Model/Paradigm</th>
<th>Origins</th>
<th>Shortcomings for OSS Modeling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data and Knowledge Model (DKM)–(Houtsma and Apers) [29]</td>
<td>EER (extended entity-relationship)</td>
<td>data model only, Horn clauses for KR make procedural specification difficult</td>
</tr>
<tr>
<td>Widom and Finkelstein’s Production system/Relational DB fusion [60]</td>
<td>Relational database</td>
<td>no explicit conceptualizations for modeling systems, lack of provision for control modeling</td>
</tr>
<tr>
<td>ODDS (Object-oriented Deductive Database System) Objects w/ PROLOG methods [37]</td>
<td>Object-oriented databases and logic programming</td>
<td>intended for sophisticated constraint modeling; no concern with control; an extension to deductive databases</td>
</tr>
<tr>
<td>Knowledge/Data Model (KDM–Kerschberg et al.) [34]</td>
<td>Functional data model</td>
<td>schema changes difficult, nonexecutable</td>
</tr>
<tr>
<td>Active Knowledge/Data Language (Active KDL)(Potter and Miller) [42], [51]</td>
<td>KDM; model base orientation</td>
<td>oriented toward Operations Research type queryable model bases</td>
</tr>
<tr>
<td>COMEX (Control Knowledge Modeling and Execution Tool) Object–production rules fusion [59]</td>
<td>expert systems development; knowledge engineering</td>
<td>no multilevel control abstraction capability, focus on smaller (single task) expert advisory systems</td>
</tr>
<tr>
<td>Commercial simulation languages [2]</td>
<td>various</td>
<td>frameworks are prestructured, nonreflective, no control abstraction capability</td>
</tr>
</tbody>
</table>

Fig. 9. A comparison of the Smart Object paradigm and prior data/knowledge fusions.

### 4.1 The Smart Object Language (SOL)

Recalling the functionality required of OSS as derived from attributes for an ideal OSS modeling and implementation environment (see Fig. 4), we wish to enable:

- Executable prototypes
- Representation of data and process constraints
- Ease of modification and prototyping
- The ability to model behavior
- Support for inferential relationships
- Reflectivity

These considerations led us to a natural language, rule based structure for the knowledge representation (KR). The Production System model [5] (chosen for our proof-of-concept language, SOL) is the best known example of this system type. The necessarily iterative nature of the development of OSS models and the emphasis on symbolic rather than computational processing make a natural language KR especially attractive [7]. The translation of the large number of heuristic, purely procedural rules into an executable model, and the verification of these rules by domain experts, is greatly eased by such a KR [58].

The highest level BNF description of SOL is shown in Fig. 10. Following the conceptual framework (see Figs. 1 and 6), the language is defined with four major sections, *Interface*, *Attributes*, *Methods*, and *Monitor*. A full BNF description of the language can be found in [7]. Additional details on the syntax pertaining to the smart object internal structure is found on the SOM www page. The sections of SOL are further elaborated on in the context of an example in Section 6.

#### 4.2 Interface

The *Interface* section explicitly defines object communication with the system. It does this through definition of the data, states, and methods that originated within the object but which can be *exported* to other objects (*public* data, states, and methods), and the data, states, and methods *imported* from other objects for use by methods in an object. Imported data, states, and methods must have been specified in an *Interface.Export* clause of another object, or an error occurs. The *accessedFrom* clause identifies for imported data the object where the data value actually resides and its name within that object. The clause is illustrated later in the context of an example in Section 6.

```
System = SmartObject [SmartObject]
SmartObject = object ObjectIdentifier
           Interface Attributes Methods Monitor
           endObject

Interface = interface
            [Export][Import]
            endInterface

Attributes = attributes
            [TypeDefinition] [DataDefinition] [StateDefinition]
            endAttributes

Methods = methods Method [Method]
          endMethods

Monitor = monitor ControlData ControlRules
          endMonitor
```

Fig. 10. High level BNF constructs of SOL.

#### 4.3 Attributes

The *Attributes* section explicitly defines data and state variables that are unique to a given object. Given correct OSS analysis [4], [47], objects partition the complex operations environment into tasks or coherent sets of decision steps, and the attributes define the data and states required for the state definition of that subset of the total operations environment.

Complex data types including array and set types can be constructed from the SOL base types which are: *integer*, *real*, *character*, *string*, *Boolean*, and *ordinal*. Ordinal types are the equivalent of enumerated variables in C and Pascal. Both state and data variables can be declared as arbitrarily complex structures of compound types.
In addition to passive data declarations, the Attributes section constitutes an integral portion of the active knowledge content of an object. The knowledge can be implemented as:

- **expressions** which initialize any data type to any type consistent value according to user defined expressions of any other system state variables
- **constraints**, which are rule references (rule based procedures) that can be triggered [41]:
  - on the satisfaction of a user specified condition
  - on first reference to the data only
  - on all references to the data
  - whenever the data is modified

### 4.4 Methods

A correctly designed smart object [4], [11] encapsulates a task or logically cohesive set of tasks drawn from a much larger hierarchical operations space (the complex operations environment). The **Methods** section is where knowledge unique to the specific real-world tasks being modeled are stored. The knowledge is expressed in production rules of the general form:

\[
\text{if condition then action.}
\]

A **condition** is an arbitrarily complex Boolean expression that semantically represents an inquiry into virtually any aspect of the state of the total system. Both private and imported data, and computations based on that data can be used as the operands in the Boolean comparisons. An **action** can be any SOL statement, such as `read`, `add`, `increment`, etc., or a reference to another method either defined in the same object, or `exported` from any other object. An expanded BNF description of SOL [7] is available on the SOM www page.

### 4.5 Monitor

The **Monitor** section together with the system production rule inference engine (not accessible to the OSS modeler) provides for the modeling of control and metacontrol [59]. The **Monitor** logic is written in SOL by the system designer to optimize the domain metamodel to the control abstractions common to a specific class of domain models [27], as developed fully in the next section. When statements involving control are selected to execute, the **manner of their execution** are determined by the **Monitor**. To enable emulation of the complex exception handling [15] frequently found in large operations environments, the monitor is divided into two subsections: **fixed rules and variable rules**. In the variable rules subsection, rules can be defined that are capable of overriding rules in the fixed control subsection. Need for variable rules, and the design of the rules, depends on the specific complex operations environment being modeled.

Although both fixed and variable rule sections are found in the Monitor section of SOL, the fixed rules section, which is common to all Smart Objects, is properly conceptualized as belonging to the **control architecture** portion of the paradigm. The manner in which SOL’s control statements, their default implementation in the fixed rules section of the monitor, and a LIFO control word stack form the control architecture is discussed next.

### 4.6 The Control Architecture

By **control architecture** [35] (H in Figs. 6 and 8), we mean the most general mechanisms by which control is implemented in an instantiation of the paradigm. The architecture also constitutes the highest level **constraints** on control modeling in any given instantiation, since only the control behaviors achievable under the architecture can be implemented. As part of the general paradigm, control inference is separate from domain, or world knowledge inference. In the SOL instantiation of the paradigm, the **functionality** of the control architecture is defined by the **Monitor Communications Statements**. To actually implement that functionality in this instantiation we decided to simply mimic the hardware control logic of a typical CPU architecture (a ‘soft’ machine) using the SOL language and a data structure typically used in operating systems for this purpose, a pushdown (LIFO) stack. The actual SOL programming for the fixed rules portion of the Monitor for the example domain is detailed in the next section. The stack elements are control records, with multiple fields for each record (see Fig. 11). The **Sender** and **Receiver** fields allow identification of the source and destination of control flows. The **Method** field indicates the rule set referenced by the **Sender**, and the **RequestState** allows control logic to determine whether to restart the inference cycle, or resume a suspended cycle with prior intermediate results intact. The **Message** field provides additional context for control decisions, as will be seen in the example in Section 6.

The **Monitor Communication Statements** implicitly load the stack record fields, and push and pop records. They implement interobject calls and returns in a fashion analogous to many traditional languages. The distinction between the SOL architecture and that of all other languages we are familiar with is that the **Monitor Communications Statements** have a privileged status and are intercepted by the inference engine during execution. Control is turned over to the Monitor—a distinct, application programmer accessible control rule set—for metalevel determination of the processing to occur for the intercepted statement. The **variable rules** section of the monitor is not part of the common control architecture, but is intended to provide the ability to override fixed rules using knowledge of exception conditions at the level of individual Smart Objects. A control system walkthrough is given in the next section of the pa-

---

**Fig. 11. Request stack of a Smart Object.**
4.7 SOL in Summary
The use of SOL in an example in Section 6 gives context and elaboration to the definition of this section. The SOL features that were engineered specifically for conceptual modeling of OSS are listed in the table in Fig. 12 (cf. Figs. 4 and 5).

4.8 Differences between SOL and OOP Languages
As we have presented SOL to various groups over time, we have observed that the significant differences between this Smart Object metamodel and “standard” object-oriented programming techniques and OODBMS’ are not always immediately apparent. The distinctions we feel are most important to the goal of OSS system modeling and development are compiled into Fig. 13. A number of researchers in the field of object-oriented languages have also suggested augmenting the ‘standard’ object model for increased conceptual clarity and support for context dependent behavior. Many of the suggested additions consist of some form of open metalevel to existing OO languages, analogous to the facility provided by the Monitor in Smart Objects [25], [53].

The instantiation of a language (SOL, in this instance) constitutes the first step towards development of a working OSS for a domain (cf. Fig. 3). In the next section, the second step in the general methodology is described: the monitor communications statements which define the high level conceptual features of the control architecture are instantiated by developing them in SOL itself in the fixed rules portion of the Monitor.

5 FROM METAMODEL TO DOMAIN MODEL
5.1 Analysis and Design of Monitor Logic
The production rules that make up the Monitor portion of a Smart Object are developed in this section. Implementation of Monitor logic is the next step in the progression from the general paradigm to a working OSS; the tailoring of the model to a more specific domain area through logic modeling of the control behavior (metaconsrol) most appropriate to the specific class of systems under consideration.

The knowledge that the domain of OSS requires models with a flexible control structure, because of the need to respond to frequent exceptions in normal operation, is embodied in the paradigm. Specifically, that requirement is provided for by the concept of a Monitor with Fixed and Variable rule sections: an explicit modifiable metacorl structure. At the level of specific class of OSS (flow type

<table>
<thead>
<tr>
<th>OSS Attributes</th>
<th>Supporting SOL Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knowledge associated with operations</td>
<td>Attributes, Interface sections support encapsulation, object structure supports high levels of data abstraction w/ associated semantics</td>
</tr>
<tr>
<td>Data</td>
<td>Monitor supports logical description of processes of arbitrary complexity in a natural language (NL) like syntax</td>
</tr>
<tr>
<td>Processes</td>
<td>Attributes, Interface syntax supports triggers</td>
</tr>
<tr>
<td>Constraints andPrerequisites</td>
<td>production system - logical descriptions of process and constraint are executable</td>
</tr>
<tr>
<td>Inferences</td>
<td>forward and backward section identifiers shift inference modality; hybrid knowledge representation provides support for generic tasks</td>
</tr>
<tr>
<td>Adaptive inferencing</td>
<td>Interface abstracts interobject relationships</td>
</tr>
<tr>
<td>Structural relationships between operations</td>
<td>new (runtime creation of objects)</td>
</tr>
<tr>
<td>Dynamic control of structure</td>
<td>Conceptually: Monitor w/Fixed/Variable rules Implementation: Call stack + Communications statements (start, resume, suspend, etc.)</td>
</tr>
<tr>
<td>Control abstraction</td>
<td>assert, retract, supersede, reinstate, updateDependencies commands; domain knowledge (Methods), control knowledge (Monitor), state variables and their semantics (Attributes) and coordination (Interface) are all interpreted at runtime</td>
</tr>
<tr>
<td>Metalevel control and knowledge facilities</td>
<td>Fixed / Variable rule distinction allows entire domain models to be reused (somewhat like frameworks in OOP) Natural language orientation to SOL syntax; expert knowledge is easily translated to SOL rule sets and reviewed by domain experts</td>
</tr>
<tr>
<td>Management of conceptual complexity through Software Engineering principles</td>
<td>Many OO analysis methodologies, and some OO design methodologies operate at a high enough level to be directly usable for system analysis using SOL; self documenting character of natural language syntax</td>
</tr>
<tr>
<td>Prototyping, maintainability, and reuse</td>
<td>Rapid development through use of frameworks; OOP type encapsulation minimizes maintenance side effects</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Construct</th>
<th>Smart Object Language</th>
<th>Traditional Object Orientation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Knowledge Representation (semantics)</td>
<td>Declarative</td>
<td>Procedural–process rather than knowledge orientation</td>
</tr>
<tr>
<td>Constraints</td>
<td>Triggered–integral part of Attributes</td>
<td>implementable but not part of the model</td>
</tr>
<tr>
<td>Control abstraction</td>
<td>Integral part of model (Monitor) easily defined and modified</td>
<td>Language dependent–Smalltalk’s metaclass, CLOS meta object protocol–neither easily defined or modified by applications programmer/analysts</td>
</tr>
</tbody>
</table>

Fig. 12. SOL constructs optimized for Operations Support Systems (OSS).

Fig. 13. SOL (an instantiation of the Smart Object paradigm) and OOP contrasted.
manufacturing systems, nuclear power plants, refineries, etc.), definition of the control problem, that is, the generation of formal specifications for the fixed control rules logic, is accomplished by domain knowledge elicitation techniques well established in both information systems analysis, and for expert systems development [4], [41].

5.2 SOL Control Syntax

Fig. 14, shows the control statement syntax of SOL, and direction of control flow between Smart Object sections for each of the statements. The detailed specifications for a control logic requirements for large power generation facilities are detailed in [44], [58]. SOL control statements have been explicitly engineered to be able to model these requirements. After the required functionality of the control statements is understood, the SOL programming that declaratively implements that functionality in the Monitor can be discussed in context.

5.3 Control System Walkthrough

In SOL, many interobject method invocations are implicit. They result from access to data which has triggered procedures attached to it in the Attributes section or derived data whose procedure is specified in the Interface section. However, no matter how control redirection is initiated, it is important to understand that all control statements are routed through the Monitor and pass control to the Monitor logic, which is simply a set of logical methods, coded in the SOL language, for dispatch of their functionality. It is convenient to view SOL’s control statements as requests for change in control flow, that will be interpreted by the Monitor control logic, and the interpretation is capable of dynamically altering in response to changes in the operating environment. Because the total control state, including interobject call history is visible to the Monitor (on the stack), the system has great flexibility in responding to exceptions, even to the point of suspending the operation of entire sections of the network of Smart Objects, and rerouting messages appropriately.

Communication between the Monitor logic and the control statements in other sections of a Smart Object is through the control stack. The status words are placed in the RequestState field of a record on the LIFO stack corresponding to the statement execution of SOL Monitor communications statements. The status word is the same as the communication statement name, that is, suspend() places the status suspend in the stack field RequestState, start() places the status start in RequestState, and so on. This allows Monitor logic to determine the status and history of recursive (nested) control transfers (see Exhibit 1 on the SOM www page).

The actual fixed and default variable rule set for the domain model is very compact, as shown in Exhibits 1 and 2 on the SOM www page. More generally, the ability to dynamically and selectively override control behavior at different points in the concept hierarchy permits:

- Different control of, and for, different classes of objects in the hierarchy. For example, a systems analysis might indicate decomposition of a system into Manager and ShopFloor objects. Manager objects might be permitted to directly access methods in their calling (supervisory) objects, while for ShopFloor objects, all such requests could be intercepted, subject to policy knowledge interpretation by an immediate supervisor object.
- Dynamic modification of effective structure in response to varying environmental situations. Workflow rerouting in the event of a machine down for maintenance, in a manufacturing environment, or the restructuring of reporting relationships in a military command and control system in response to casualties are examples of OSS need for dynamic, radical restructuring.

Once the control logic of the domain has been analyzed and implemented in the Monitor, a framework exists that can be used in multiple specific physical sites of the domain. Completion of a working OSS requires only the population of data structures and the definition of variable rules (if any) to accommodate site specific information and control. This is illustrated by the example in the next section.

Fig. 14. SOL communications statements.
In this section, we present portions of a subsystem from a much larger prototype OSS for a nuclear power plant [7], [58], to illustrate salient features of SOL and the underlying paradigm. The example provides elaboration of the SOL definition of Section 4.

The central methodological concept of the paradigm is the partitioning of the modeled operations support system into a domain component, and an applications component. The domain component is modeled by a set of objects that represent control and structure elements common to a class of OSS, such as chemical process plants, power plants, a manufacturing operation using a certain type of assembly flow, etc. The domain component constitutes a potentially reusable core for a given class of model. As a result of the inheritance property of objects within the paradigm, if a common class domain structure requires minor revision, a model specific object can be created that inherits from a "standard" object and has selected rules or attributes overridden at the instance level to match the specific circumstance. The applications component is unique to each model instance. In this section of the model resides the knowledge specific to an actual site, the Georgia Tech Research Reactor, for example. Standard knowledge engineering techniques such as those used in the development of expert systems are used to elicit this information [23].

One of the consequences of the use of an object-oriented conceptualization is that many of the methodologies and techniques developed for object-oriented analysis and design for OOP environments [11], [12], [4], are useful in whole or part for development using Smart Objects. This is so because most such techniques deliberately stay at high levels of abstraction, divorced from coding details as far into the development process as possible. Our experience indicates that analysts familiar with OOP environments are quickly comfortable with the SOL environment. (However, as anticipated, OO programmers are less comfortable with the declarative nature of logic based programming in SOL.)

The portion of the nuclear power plant OSS prototype from which the example is taken is shown in Fig. 15. The target environment has been partitioned into smart objects, some of which represent both domain knowledge and application knowledge. The domain objects, Procedure 2000, Procedure 4000, Procedure 7245, and Procedure 7246 encapsulate knowledge that will remain constant over a class of models, in this case nuclear power plants. The objects instantiate procedures mandated by the Nuclear Regulatory Commission for all domestic nuclear power plants, and could be substantially reused in models of different specific sites. By contrast, the application objects, Executive and Personnel, encapsulate knowledge and information that is specific to the particular site being modeled.

It is the Executive object that integrates the various operations together into a support system that provides a global view of system operations. It accomplishes this by monitoring the relationships between the highest level system control abstractions (state variables). These control abstractions are explicitly dependent (through SOL statements) on the performance of various procedure objects. Due to considerations of space, only the shaded subsystem of the diagram consisting of the objects Executive and Procedure 7245, will be discussed in the narrative that follows.

The knowledge associated with Procedure 7245 is shown below in Fig. 16; the procedure involves the measurement of the reactivity in the reactor core. This measure, called the Shutdown Margin, must be equal to or greater than an established threshold in order to ensure that the reactor can be

<table>
<thead>
<tr>
<th>Procedure 7245—Reactor Shutdown Margin Determination</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>References</strong></td>
</tr>
<tr>
<td>Frequency</td>
</tr>
<tr>
<td>Constraints</td>
</tr>
<tr>
<td>Prerequisites</td>
</tr>
<tr>
<td>Data</td>
</tr>
<tr>
<td>Condition</td>
</tr>
<tr>
<td>Constraints</td>
</tr>
<tr>
<td>Condition</td>
</tr>
<tr>
<td>Violation</td>
</tr>
<tr>
<td>Violation</td>
</tr>
</tbody>
</table>

Fig. 16. Knowledge associated with Procedure 7245.
safely shut down at all times. Note the similarity between this procedural knowledge and similar sets of procedures in manufacturing, chemical process industries, commercial fleet maintenance of vehicles, and other logically closed processes. The control architecture and possibly the overall object architecture could be reused for these domains.

In the example that follows, the SOL code required to implement the two objects, Executive and Procedure 7245 is displayed and amplified by narrative where appropriate. Variable rules for the Monitor of Procedure 7245 are shown in Fig. E5 on the SOM www page.

6.1 Examples of SOL Usage

The interface section of an object defines an object’s communication with other system objects (see Figs. 17, 18, and E6) (on the SOM www page). This communication is separated into two parts: export and import. The export part consists of domain (or application) data/state values and methods which may be used or accessed by other system objects.

The import part declares application data/state values, and methods of other system objects that are needed by an object to perform its given task or which may affect decisions made while carrying out a task within the object. This section captures the data flow and control flow relationships between objects and aids the developers in understanding the complex relationships that exist between system activities.

The Attribute section of an object (see Figs. 19 and 20) includes all domain and application data and state values associated with an object. A state represents a certain condition or property of an object that holds true at a certain point in time. A state, generally speaking, is an abstraction of a data value or set of data values of a given object’s attributes. States are used by the Monitor of an object to make decisions concerning what to do next. In addition, states may be exported to other objects which contain the appropriate knowledge to respond to the meaning of a given state. States are an important concept in the model that can be used to abstract the status of an object at a local level, and at a higher level they abstract the status of several objects within the system.

Attributes and states may have associated rules that derive or infer the value or values of an attribute when needed, specify constraints on the values of an attribute, and specify the actions to be taken when an attribute value is inserted, deleted, or accessed.

The Methods section of an object (see Fig. 21) consists of a logical grouping of condition-action rules which perform a given task for an object. Thus, application rules are stored in the method section to represent application processes as well as in the attribute section to represent rules associated with application data and states. The rule-based paradigm or production system model of computation [23], [5] is chosen to represent domain/application rules (as well as control rules) of an object because of its ability to naturally represent well understood chunks of knowledge in a problem domain.
These chunks of knowledge are associations among patterns of data and the actions that are taken when configurations of data in the problem space match these patterns. These rules fire when their condition parts are satisfied. Just as in procedural object-oriented programming environments, encapsulation of knowledge dramatically increases the potential for reuse. The measurement technique for Shutdown Margin and its status are contained completely within the \textit{Attributes} and \textit{Methods} sections of object \textit{Procedure 7245}. The availability and names of personnel are contained completely within site specific object \textit{Personnel}. Data and procedures for either of these objects can be changed without effect on \textit{Executive}, whose methods are responsible only for coordination and scheduling of the tasks represented by the process objects. Process activities are abstracted as states, which remain constant even under drastic modification of the task procedures.

6.2 From Domain Model to Working OSS

To finalize the transition from paradigm to working OSS, it is necessary only to populate the domain model with site specific data and control exception conditions. When developing OSS’ using the Smart Object paradigm the environment specific, but unpopulated objects are usually provided for in the domain framework. In the case of the example just discussed, site specific information would be used to complete definition of objects \textit{Personnel} and \textit{Executive}. In the process of successively instantiating the paradigm, we have developed both a working, site specific system, and a domain specific framework, optimized for the environment it is intended to model and simulate, and highly reusable in a variety of similar installations. The control model developed in this
exercise would also be appropriate for any domain where activities are constrained by precise procedures. Open systems [46] would require a more flexible and sophisticated metalogic and control architecture.

7 Conclusions

In this paper, a new class of data/knowledge representation, the Smart Object paradigm, has been developed, initially to assist in the modeling, design, and development of Operations Support Systems (OSS) for complex operations environments. The paradigm integrates data/knowledge abstractions from semantic data models with concepts from natural language rule based models of knowledge representation. As a result, it can declaratively represent data structures and task knowledge, and multilevel control abstractions. It differs from previously proposed data/knowledge paradigms in its ability to produce an executable final product, in its stress on the dynamic aspects of the modeled environment, and in the ease with which metacontrol of knowledge inference behavior can be implemented and modified.

Metalevel control in production systems has been proposed by the artificial intelligence community, to enable adaptive system behavior [45], and by the knowledge and data engineering community to ameliorate maintenance problems in large expert systems [32]. However the segmented object architecture of the Smart Object paradigm provides metacontrol capability without the use of multiple language paradigms (one procedural, one declarative), a solution we feel is both conceptually consistent and more maintainable.

The central concept of the paradigm is that of the “smart object,” an encapsulation of task knowledge, control knowledge, data, and procedures, which can used to model complex operations environments. Control is modeled at a metalevel so that control is determined as the system reasons about control situations, as they occur, using its control knowledge. The current instantiation, SOL, uses a natural language syntax production system for its knowledge representation. In practice, a system is modeled and designed as a network of smart objects between which data and control information are passed. A logic engine makes inferences based on the knowledge contained in the network, and has the ability to modify the system state, structure, and control flow in response to environmental information. Some of the objects represent a potentially reusable core of “domain” knowledge common to a class of modeled environment, and others capture “application” knowledge, unique to a specific modeled instance.

7.1 Current Limitations

Some aspects of the paradigm are not yet fully developed. Two areas of notable interest are a graphic notation and programming interface, and a formal method of validation of correctness of the designs. A graphic interface is currently under development. As regards correctness, the potential within the paradigm for heuristics, defeasible and higher order logics greatly complicates rigorous logical proof; however, we believe formal specifications using tools such as VDM [33] or Z [16] should be as applicable to SOL as to any other mathematically complete language. Additionally, a truth maintenance subsystem for dynamic checking of consistency in the knowledge base could be added to the development environment.

7.2 Future Research

The paradigm has proven richer than anticipated, with applicability beyond its original intent. Our exercise of it for OSS development and later for other nontraditional database applications requiring active knowledge representation, have suggested a number of directions for future research. In addition to the graphical interface, projects either proposed or underway are:

- A redesign of the fixed control data and rules in the Monitor section of a Smart Object in order to handle multiple threads of control and active database modeling capability.
• An instantiation of the paradigm for use in an open (not fully constrained) domain, such as office information systems [46] or workflow management systems, where the metacontrol aspects of the paradigm can be more fully utilized.

• An instantiation of the paradigm for the modeling of expert systems [36]. A knowledge (rule) base partitioned as suggested in the paper yields superior performance to an undifferentiated rule base of equivalent total number of rules [38], and the hierarchical object structure efficiently models interacting tasks in complex knowledge based systems.

• Development of a formal methodology that provides guidance for the use of SOL in the modeling and design of operations support systems.

ACKNOWLEDGMENTS

The authors are indebted to Sham Navathe, David Spooner, Benjamin Wah, Sandeep Purao, Rajiv Kishore, Dan Turk, and the anonymous referees for their valuable comments on earlier versions of the paper which have helped in improving and sharpening the presentation of the paper.

The work of Vijay K. Vaishnavi was partially supported by a grant from the Research Program Committee of the College of Business Administration, Georgia State University, Atlanta, and a grant from the Georgia Research Alliance. Exhibits and appendices to accompany this paper are available at URL: http://cis.gsu/~vvaishna/som.html (SOM www page).

REFERENCES


Vijay K. Vaishnavi received the BE degree (with distinction) in electrical engineering from Jammu and Kashmir University and the MTech and PhD degrees in electrical engineering from the Indian Institute of Technology, Kanpur. After receiving his PhD degree, he was a postdoctoral fellow in computer science for two years at McMaster University, Ontario, Canada. Dr. Vaishnavi is currently a professor of computer information systems at Georgia State University. He is also research director of the Consortium for the Management of Emerging Software Technologies. His current areas of research interest include software engineering, knowledge modeling, and data structures. He has authored and refereed numerous papers in these and related areas. His papers have appeared in IEEE Transactions on Software Engineering, IEEE Transactions on Computers, Communications of the ACM, SIAM Journal on Computing, Acta Informatica, Journal of Algorithms, and several other major international journals and conference proceedings. Dr. Vaishnavi is a senior member of the IEEE. He is also a member of the IEEE Computer Society, IEEE Technical Committee on Software Engineering, the ACM, and SIGSOFT.

Gary C. Buchanan received the BS degree in computer information science from Troy State University at Dothan, Alabama; the MS degree in computer information science from the University of Alabama at Birmingham; and the PhD degree in information systems from Georgia State University, Atlanta. He is currently chair in the Department of Computer Information Systems at Troy State University. His current research and teaching interests include knowledge modeling, client/server development environments, and business process reengineering. His papers have appeared in Expert Systems Review with Applications and in the Proceedings of the International Conference on Systems, Man, and Cybernetics. Dr. Buchanan is a member of the Association for Computing Machinery.

William L. Kuechler Jr. received the BS degree in electrical engineering from Drexel University, Philadelphia. He entered the doctoral program in computer information systems at Georgia State University, Atlanta, in 1993. He has substantial experience in the software engineering and information systems fields, in both technical and technical management positions. His commercial system development experience ranges from large COBOL systems managed with waterfall methodologies through object-oriented projects in C++, which used iterative incremental methodologies. His research interests include abstractions and methodologies for the design of complex systems, and the cognitive aspects of system development and software reuse. His papers have appeared in the Proceedings of the Hawaii International Conference on System Sciences (HICSS) and other conference proceedings.