American manufacturers must use computers to automate production if the US is to compete successfully in an international economy. Researchers around the world spent about $40 billion in 1990 trying to solve planning and control problems in industrial automation, with about $20 billion of that going to four techniques: computerized manufacturing systems, computer numerically controlled (CNC) systems, robots, and flexible manufacturing systems. Much of the money invested in these technologies was spent on developing the most crucial part of the system: the control software.

We describe information and decision processes in terms of three fundamental concepts: task decomposition, allocation, and updating. Our object-oriented, logic-programming paradigm maps these concepts to computational elements, providing the basis for a powerful programming environment.

We propose using a formal method to develop control software for flexible manufacturing systems. Formal methods are math-based techniques that make it easier to describe software specifications for complex systems and to support software verification and testing. They help to reveal ambiguities, expose errors, communicate complex ideas, and prove the correctness of system properties. They are especially important when there must be a clear boundary between specification and implementation. Formal methods are being used in major software projects such as IBM's CICS, Praxis Systems' Case project, and the Portable Common Tools Environment.

Our formal method is based on the Semantic Unification and Logic (Semlog) environment, a typed object-oriented programming environment. Our approach tries to minimize the effort needed to transform the manufacturer's problem description (with its parts, machines, and objects) into the programmer's solution plan (with its programs and data). The representation scheme facilitates abstract and high-level descriptions as well as automated reasoning by using primitives representing the domain elements (lathes, engines, and so on) and their interrelationships (for example, a manufacturing schedule). Our objective in this article is not to elaborate on tools for simulating manufacturing processes, but rather to describe the programming language constructs that facilitate software modeling and the development of decision-making algorithms in a flexible manufacturing systems.
Planning and control in an FMS

A flexible manufacturing system, or FMS, is a flexible collection of communicating groups of modular, automated material handling devices and numerically controlled, interchangeable machine tools, all connected by flexible communication links and material-handling systems and integrated by a hierarchical network of computers. These devices and tools can simultaneously process medium-sized volumes of various part types. FMSs have been used to improve quality, productivity, use of capital, and responsiveness, and to reduce material handling, labor costs, and inventory.7

An FMS can have flexibility along several dimensions:8

- The machine: making the changes needed to produce a given set of part types.
- The process: producing a given set of part types, each possibly using different materials, in several ways.
- The product: producing a new set of products economically and quickly.
- Routing: handling breakdowns while continuing to produce a given set of parts.
- Volume: operating profitably at different production volumes.
- Expansion: building a system in modules and expanding it as needed.
- Operation: interchanging the order of operations for each part type.
- Production: producing a wide universe of part types.

Such a categorization is important to designing a representational and reasoning scheme that exploits an FMS's intrinsic capabilities.

An FMS is an adaptive, dynamic system in which a wide variety of jobs are continuously and randomly introduced. These jobs must be broken down into operations, which then have to be scheduled on various machines. Completed jobs must be removed from the system with minimal, if any, human intervention.

The computers in an FMS carry out different levels of planning and control using heterogeneous, logically integrated, intelligent, autonomous, and spatially distributed processors that share a common goal. Many hardware and software processors are distributed over several distinct hierarchical levels. When left on their own, the processors can make decisions over a wide range of informational and decision tasks without intervention either from people or from higher levels of the system. The data and knowledge bases are also distributed, so that each processor has its own memory. At any instant, the total information base of the system is partitioned among the processors' individual databases. An intermodule communication system sends messages for instructions, control, and database updating. Processors coordinate activities by exchanging messages. Each processor responds deterministically to a well-defined set of messages and can send messages to other processors in order to handle its own incoming messages.

Hierarchy is a natural way (and often the only way) to control large, complex systems. (This view is not universally held. For example, Rana and Taneja claim that hierarchical approaches cannot deal with the combinatorial explosion of complexity presented by very large systems, they are inflexible and difficult to expand, and they introduce coordination problems.) The FMS decision-making process spans a control hierarchy with five levels: the equipment (tool), the workstation, the cell, the shop, and the facility.11

The system decomposes the goals of the higher levels into commands that are executed by the lower levels. Feedback from lower levels is passed back to the higher levels, but at a decreasing rate (since this information is abstracted into more complex constructs). At the upper levels the FMS makes decisions that will apply over a long time, but this "time horizon" progressively reduces down the levels.

Individual machines and robots. The tools at the lowest level are responsible for activities such as matching, measuring, handling, transporting, and storing. Each must be able to

- exchange messages with the workstation and the cell controller;
- allow access to machine and controller states;
- receive input from sensors, and
- send output signals to motors, solenoids, valves, and other devices.

Process control information lies at this lowest level in the hierarchy, where decision making must adapt to the environment. A controller should be able to cope with contingencies, including occasional failures from such causes as faulty parts, broken tools, and robot drift. An intelligent and autonomous controller must exhibit rudimentary abilities in error detection and recovery.

Planning at this level involves the scheduling of individual operations, such as moving, gripping, measuring, and matching. The time horizon is seconds or hours, and the processing is done in the FMS computer.

Workstations. At the workstation level, the system manages various manufacturing operations (setup, equipment tasking, and takedown) and resolves contingencies and conflicts. This level typically includes machines for milling, inspection, and material handling. The time horizon is minutes or hours, and processing is done in the FMS computer.

Cells. A manufacturing cell is a group of machine tools and associated material-handling equipment that is managed by a supervisory mainframe or FMS computer, called the cell host. Cells behave as independent units and need to be integrated to form an FMS. They constitute the middle level, and their principle decision-making function is to coordinate and manage resources. Toward this end, the cell host schedules the flow of jobs in the cell, maintains a database about pending operations, executes the software for gracefully degrading the system in the event of failure, and instructs the individual elements as to the tasks each must perform. The time horizon at this level is hours or weeks.

The shop. This level manages tasks and
allocates resources. The decision and control functions exercised by the shop-level supervisory system are similar to those of a cell, but it also dispatches material-handling equipment, monitors cell conditions, and collects data. The time horizon is weeks or months, and processing is done in the mainframe.

Much operations research modeling has been concerned with scheduling and sequencing issues at the shop level; few, if any, models have been suggested for planning and control at other levels. However, researchers have suggested tools based on petri nets for modeling cell-level interaction and for designing a shop-level operating and control system.

The facility. As the highest level in the FMS hierarchy, the facility deals primarily with information management, manufacturing engineering, and production management. It handles the premanufacturing phase of planning, including material requirements planning, master production scheduling, and batching. These plantwide functions typically set the overall goals and targets for the long term. The time horizon at this level is months or years. Processing is done on the mainframe.

Generic planning and control functions. Each level of the hierarchy is associated with an object that encapsulates the planning and control functions at that level (see Table 1). These functions fall into three generic categories:

- **Task decomposition.** The facility-level task of manufacturing an engine can be split at the shop level into the tasks of casting, machining, and assembling. At the cell level, the tasks are further broken down into such tasks as drilling, boring, and finishing. Finally, the task description at the individual machine or robot level might include loading, moving, and gripping. Each level has its own set of task symbols that act as a language alphabet for that level. To support task decomposition in an FMS control program, we need program language constructs (described later) that facilitate hierarchical data and program definition schemes, as well as primitives for coordination.

- **Task allocation.** An FMS must assign jobs to specific machines or cells. Dynamic allocation can be very efficient but it is hard to program, especially at the facility and shop levels, because changing contingencies are an intrinsic part of the environment, and a great deal of skill is required to identify and eliminate errors. In task allocation, the key is to balance the workload. This minimizes the differences in time required for jobs assigned to different units, and ensures that all the work for each batch is in fact assigned to some machine. To allocate tasks, FMS control programs need to support incremental specification and the encapsulation of data and programs.

- **Task updating.** The FMS must maintain a database of pending and completed tasks. Thus, when a robot has finished cutting a gear, all the parent manufacturing cells must be notified that the robot is free and can begin the next allocated task. Task updating must be robust to maintain the consistency of the underlying database. For example, a robot cannot be declared free unless the job that has been scheduled on it has been completed. To support task updating in a dynamic manufacturing environment, the FMS must have an adaptive control strategy based on the exchange of information and messages.

The information flow in an FMS not only coordinates conversion steps (the steps that change raw materials into finished goods) but also provides the feedback necessary for improvements. Thus the control software must be based on a model that supports the highly distributed, complex, and layered nature of FMS information sharing and decision making.

### Knowledge representation requirements

All three types of FMS planning and control activities—task decomposition, allocation, and updating—have a common thread: They are completely controlled by computer programs. Product and process specifications exist as computational procedures that are fed into these programs. Stored specifications enable factories to produce identical parts on different machines in different places. These specifications must anticipate every possible contingency and solve every potential problem with the product and the process. Thus, the basic conceptual approach to designing FMS control systems must be to support the distributed, multilayer, and interconnected nature of FMS decisional and informational processes with flexible, easily maintainable, and modular software.

Suri and Whitney propose using decision support systems and knowledge-based systems to design FMS control software.

Knowledge-based systems differ from usual software systems in that they explicitly encode domain knowledge. Their linguistic and procedural characteristics are also important: their power to express concepts, and their ability to manipulate and reason with such knowledge.

Effective representation of domain-specific knowledge is a critical step in build-
ing successful knowledge based systems. When we examine the domain-specific representa-tional and computational aspects of a typical FMS control system, we find new representa-tional needs beyond the current capabilities offered by multiparadigm-based systems such as the Knowledge Engineering Environment and the Lisp Object-Oriented Programming System. An FMS methodology must support a variety of requirements under uniform semantics. It must distribute the representation of procedural knowledge among various entities in a taxonomic hierarchy, and support the inheritance of this knowledge. It must also represent changes in dynamic systems using a formal system with well-defined semantics, such as lambda calculus or first-order logic. In short, the knowledge representation framework should allow a world view in which the system's functional relationships can be described in the same way they are perceived by program-mers and manufacturing experts. The representational methodology should support procedural, constraint, spatial, and tempo-ral knowledge.

To formally specify a conceptual description of the decision model, we must translate the model into the paradigms and constructs of programming languages: objects, types, and functions (see Figure 1). Thus, to effectively model an FMS control system, a knowledge-based system must

- provide primitives for easy and natural modeling of domain objects, and support multiple inheritance through a lattice structure of classes and types;
- be strongly typed to allow static type checking, which improves software reliability;
- be expressive in modeling changes in the manufacturing environment, and semantically clear; and
- be able to represent arbitrary relationships among object types, and deductions based on these types.

**Object-oriented representation.** Knowledge representation in an advanced computational system such as ours constitutes a program definition. When a developer looks at a program definition from a linguistic perspective, an immediate question is its expressive power and adequacy for a given domain. In the last two decades, researchers have introduced a broad variety of integrated knowledge-representa-tion languages such as first-order logic, frames, and production systems. For example, the developers of frame systems decided to synthesize declarative and pro-cedural approaches by emphasizing an object-oriented representation.

An object-oriented representation offers an integrated view of abstract data types and generic functions. The processing model is characterized by communicating objects that send and receive messages. Each message is itself an object, with its own behavior-al protocol (the set of messages it can respond to) described as part of its specification. By collecting objects with the same protocol in one class, object-oriented systems structure a domain (object) world through a class system. Classes are usually ordered in generalization hierarchies, so we can define classes of objects as extensions or restrictions of each other through inheritance mechanisms. Each subclass inherits the properties of the superclass, which has the nature of an ordered list representing a queue. Inheritance allows us to define new units of software according to their similarity to existing units, provides a form of abbreviation, and improves software organization. If we use a single inheritance model to describe the class structure, a class can be the heir (inherit from) a single class. In the case of multiple and repeated inheritance, a class can be heir of more than one class and more than once to the same class. Figure 2 shows an example of objects and their communication.

Besides inheritance, object oriented sys-tems (including ours) have two other defining characteristics: encapsulation and poly-morphism. Encapsulation allows systems to be modularized on the basis of their data structures, and objects can be described as implementations of abstract data types. The specification of an operation, which is visible, is clearly distinct from its implementa-tion, which is hidden. Encapsulation intro-duces a new unit of reusability, the class, and facilitates information hiding, since objects are only visible through their interfaces.

Polymorphism lets an object accept different kinds of arguments and respond accordingly. By permitting an object to have more than one type, polymorphism promotes fewer, more reusable operators. Commands in an FMS control environment are polymorphic, since they can accept different combinations of parameters.

Object-oriented programming is character-ized by

- universal polymorphism, where one implementation of an operator handles all objects; and
- ad hoc polymorphism, where an oper-ator has multiple implementations (the function executes different code for dif-ferent types or combinations of types).
Universal polymorphism can be either parametric or inclusive. In parametric polymorphism, a function has a type parameter that determines the argument type for each application of that function. In inclusive polymorphism, an object can belong to many different classes through the use of subtyping.

We can model an engine-manufacturing plant using objects such as engines, facilities, processes, and inventories (see Figure 3). The class of engines leads to such subclasses as marine engines, automobile engines, and truck engines. Each engine can be subdivided further into subclasses, including engine blocks and crankshafts. We can also describe an engine with attributes such as horsepower, revolutions per minute, and model number, which are inherited by each engine subclass.

Similarly, we can subclassify facilities into machining, foundry, and quality control, and further subclassify machining into boring, milling, and drilling. A process can be subclassified into various assemblies and testing. Inventories can be subclassified as inventories at various centers. Each subclass representing an operation specifies a sequence of operations as its attributes and also inherits the properties of the superclass. Objects in the FMS world and objects in the program environment are related one to one.

An object is characterized by the type of message it responds to. Objects respond by executing a procedure or process. An Engine object could respond to an Update message by updating its current state; a Machine object could respond to a Load message by loading a waiting job. The scheduling function can be encapsulated in a Shop-Scheduler object that can schedule jobs on a real-time basis.

In an FMS control system, an object-oriented approach increases the representation power and robustness of the software, because the underlying data management system allows new types of entities and relationships to be easily defined according to the needs of the task. The FMS control process is typically thought of in terms of events and event handling, which have a natural correspondence to the way object-oriented systems model objects and messages. This approach provides a rich descriptive system for constructing FMS control software because of the easy transition from the design metalanguage to a formal (computable) specification.

**Typing.** Mathematically, a function can only be applied to objects in its domain. Programmers generally classify the objects in a domain into interrelated classes called types, differentiating the objects according to their attributes and applicability as arguments to functions. Static type checking tries to ensure that domain incompatibility (applying a function to objects outside the domain) does not occur in a program description; in other words, that the function is going to act on domain compatible objects, which in turn helps ensure the program's reliability and integrity. Statically typed languages support type checking at compile time or before execution. Dynamic type checking tries to ensure that domain incompatibility does not occur during program execution. All sound programming languages that support large-scale systems development, such as Pascal and Ada, incorporate typing.

In FMS control systems, enhancing type checks during execution by type checking before execution results in two main advantages. The first is efficiency: A procedure might be executed many times, but it needs to be type checked only once. Systems can also use type information to improve the efficiency of compilation. The second advantage is effectiveness: Type checking helps to pinpoint programming errors before execution and thereby simplifies program testing and debugging. Typing improves program readability by making explicit the data types used by the programmer. On the other hand, the discipline imposed by type checking manifests itself as constraints on the programmer's freedom of expression. In large software systems where reliability, integrity, and efficiency are major issues, the advantages of typing far outweigh the disadvantages.

In a typed language, the type of every function expression can be determined from the type of its arguments and the type of the function. Whenever a function and its arguments are evaluated, the result can only be of a specific type. Many object-oriented languages and programming environments, such as Smalltalk, KEE, and LOOPS, do not incorporate the concept of static types to detect domain incompatibilities. Without type information, nothing in the system guarantees the integrity of a knowledge base built in these systems.

Since the concept of typing is close to the concept of classes, multiple inheritance can be realized through typing. The types in a system are related to one another by subtype relationships. We say that a type $\tau$ is a subtype of type $\sigma$ when all the values of $\tau$ are also the values of $\sigma$. In other words, type $\tau$ inherits the attributes of type $\sigma$. This kind of inheritance imposes structure on a collection of related types and thus reduces the complexity of the system specification. The subtype relationship permits the flexible use of objects of one type for another type. Whenever an object of type $\sigma$ is permitted, the language allows for an object of type $\tau$, if $\tau$ is a subtype of $\sigma$.

Different subtyping rules apply for different type structures. Using a set-theoretic approach, $\tau \subseteq \sigma \iff \forall x. (x \in \tau \Rightarrow x \in \sigma)$.

Our programming language, Semlog, supports strong typing: In other words, in any context in which an object is used, the object type must agree with the type expected in that context. Strong typing ensures the safety of expressions, provides a form of automatic documentation, and supports generality and polymorphism. It also increases system flexibility by enabling late binding.
A strongly typed system also permits the use of automated type-inferencing tools, which are theorem provers for a collection of type constraints. There are two sets of type constraints:

1. The assignment set contains associations of an identifier with a type expression, denoting that bindings for the identifier are members of the set denoted by the type expression.

2. The restriction set contains subtype or inclusion relationships among type expressions. The latter contains the constraints that define rules among parameters.

Given a collection of known type constraints, a type inference system (as is supported by Semlog) can answer two kinds of questions. First, it can determine whether an expression is well-typed, meaning guaranteed to produce compatible values and not permit illegal operations such as the addition of an integer and a string. Second, a type inference system can deduce unknown type information about identifiers and subexpressions. These systems are usually designed to deduce the principal or most general type, alleviating the user from specifying redundant type information. The use of type variables and their subset conditions makes such systems flexible, because only the most general type is determined at compile time. The actual type at execution is a substitution instance of the most general type.

For example, a Record object called Planner has a field embodying the function Planning, which is defined over two type arguments: Orders with type ConfirmedOrder, and Joblist of type JobType. This produces a machine-job sequence of type Plan. If the message sent to Planner has as its argument a list of Orders with type ExpectedOrder, the system will refuse to evaluate the function. However, if the argument type is ConfirmedOrderWithAdvance (a subtype of ConfirmedOrder), the system will evaluate the function and return a record object with type Plan.

One such type-inferencing tool improves the quality of understanding of commands in IBM’s Distributed Data Management architecture.

Functional languages. Functional languages carry out computation entirely by evaluating expressions. They are declarative languages whose underlying model of computation is the function rather than the relation. Thus, logical languages such as Prolog are also functional, though relational in type. A program written in a functional language consists of independent functions, each mapping its argument types to its result types. It uses recursive function definitions and sequential compositions to build higher order functions.

In functional programming, an expression’s value is determined solely by the value of its constituents. Thus, if the same function and arguments occur twice in different contexts, they denote the same value. A language that supports this property (known as referential transparency) is referred to as a purely functional language. This keeps the language’s semantics simple and obvious, and helps to build FMS control software systems by simply aggregating subsystems.

Most functional languages have evolved around Lisp, which in turn grew out of lambda calculus, a theory of functions. The first functional programming language was Church’s lambda calculus, a syntax for terms and a set of rewriting rules for capturing the behavior of functions. Relational languages output a set of values as the result of applying a function—instead of a single value, as do Lisp and other purely functional languages. When the substrate of a programming language is a formal mathematical system such as lambda calculus or first-order logic (as is the case with Semlog), the semantics of the language are unambiguous. This lack of ambiguity and the power of formal specification in FMS control software is critical for reliable operation. As ad hoc amalgamations of disparate language paradigms, KEE and LOOPs have very complex semantic descriptions.

FMS environments can be described in terms of functional paradigms. In the FMS context, manufacturing processes that alter objects as a result of applying processes can be viewed as applying functions. For example, the function Drilling maps objects from a domain of type MetalBlock to objects in the domain of type MetalBlockWithHole (a subtype of type MetalBlock). Similarly, a high-level information-processing function such as Planning could be built out of a sequential composition of lower order functions such as FindSequence, Schedule, ReserveMaterial, and so on. Each lower order function in turn could be built out of more primitive functions. Primitive functions such as Schedule need not be implemented in a functional language; they can be programmed in an applicative language such as Fortran embodying any suitable operations-research algorithm. Higher order functions access these lower order functions through external (and possibly remote) procedure calls.

This expressiveness allows us to model the manufacturing process in an FMS environment through functional composition, and perform several types of reasoning at the domain level. This ability to express changes in the manufacturing environment is a key factor: For example, we can easily compute alternative production paths.

Logic programming. Our approach uses logic programming to model constraints at the domain level. For example, we can check machine availability or material requirements prior to scheduling a job.

In declarative languages (like Prolog and many logic-programming languages), no implicit state of computation exists. They perform state-oriented computations by carrying the state explicitly (as in recursion), and they emphasize programming with expressions or terms.

Logic programming is based on a resolution theorem prover. Given a set of logical formulas, a theorem prover can determine whether a contradiction exists. In the logic paradigm, a program is a collection of formulas in predicate logic containing a theorem to be proved. The logic programmer is less concerned with the process of theorem proving than with en-
suring that the logic formulas in the program are a true reflection of the problem domain. The advantage of a logic program is its separation of concerns. We can specify a program’s logic and control separately: once defined, the same control can be used by many programs. Logic programming allows a declarative representation for entities and their relationships. This nonprocedural specification increases the reliability and robustness of FMS control software because the interpretation and modification of these objects is as simple as possible.

Horn clauses in Prolog and Semlog can be read as implications, which are universally quantified by the variables in the clause. From a knowledge-engineering and conceptual-modeling perspective, Prolog’s most important assets are its declarative representation of knowledge, its ability to represent arbitrary relations among objects, and its deductive capability.

While Prolog has several characteristics that make it suitable for programming knowledge-based systems, it also has several drawbacks. In the logic interpretations of Prolog, first-order terms that are not variables appear as constants or functions. These functions are treated as record constructors and are never evaluated. In Prolog, the only way to organize data is through the use of functional first-order terms, called functors.

Using terms to represent records has limitations. First, the interpretation of the argument’s position is not transparent to the user. Also, the same functor can be used for two different record structures with different numbers of arguments. In Prolog, functional behavior has to be simulated by rules and facts, and their ordering in the database plays an important role in determining how they are interpreted.

It is much easier to design systems for FMS control when vocabularies in the programming language relate directly to the real world. Conceptual models view the world in terms of entities or objects that have associated descriptions and that are related to each other in meaningful ways. Many languages have been developed to express this information in a conceptual model. These languages provide facilities for forming arbitrary relations among objects, which we must be able to represent in order to model the informational and decisional processes in FMS control. However, these languages generally lack facilities for reasoning about relations. In other words, these languages have no facilities for representing rules about the domain of relations.

Semlog

Semlog is a high-level, strongly typed language that allows for programming with relations. However, unlike logic languages such as Prolog, where the arguments of a relation are restricted to variables and first-order terms, Semlog allows its predicate arguments to be expressions. Since it is not necessary to declare inheritance specifically, Semlog implements multiple inheritance through type statements and type-object lattices, and deduces subtype relationships from type structures. Inheritance, thus depends only on the structure of objects. This is in contrast to Smalltalk, where classes are matched by names and where the inheritance relation between classes is explicitly declared. Thus, inheritance as a special relation among objects is separated from the logical process.

Semlog supports dynamic binding (associating an implementation with a feature at runtime). This guarantees the type safety of an expression at compile time but chooses an implementation at runtime. Dynamic binding increases flexibility but at the cost of system performance.

Types. An abstract data type is a description of the services (features and operations) of a data structure without the implementation details; it is the public description of a class. In Semlog, an abstract data type is defined by an expression of type Word. Semlog also allows recursive type definitions. The subtyping relationship is expressed as an Is-A relationship in the type definition expression. The Is-A relationship is distinct from type relationships, which are defined over other types and on which the system can perform inferences based on unification and resolution refutation (as in Prolog). Subtypes inherit all the fields of the supertype. Types described as abstract types exist solely to describe a property set. Types can also be classified as passive or active. The former has only data structures, while the latter has both data structures and functions.

The Relation construct connects two or more types and can be asserted in a database. The type of every relation in Semlog must be specified prior to its use in a literal. For example, P(a, [b,b]) states that the type of the relation P is a, [b,b], that is, P is a binary relation (or predicate) between expressions of type a and the record type [b,b]. These and similar assertions on types enable the system to infer and deduce other relationships between types. Type constructors, including ListOf, SetOf, functions, and variant records, allow us to build composite types from primitive types. We can use Boolean, real, integer, time, and time-interval expressions to define composite types. The sidebar on page 60 describes other type definitions in a typical manufacturing system.

Records. Semlog uses Record types to model classes, and record instances to model class instances. Most composite types and all objects are described as records. Like ListOf, Semlog’s record structure is a type constructor. Records can have both passive and active fields; the latter contain functions (the counterpart of Smalltalk methods) and are defined as such (this is not possible in Prolog).

Semlog sends a message to an object with arguments by selecting a field in a record that embodies the function and applying the function to the arguments. The important difference with languages such as Smalltalk, LOOPS, or KEE is that Semlog’s arguments have specific types. Its functions are applied only after types are checked, and it can substitute any object with an appropriate subtype for a given type.
Type definitions in a typical manufacturing environment

Type Dwg-Status = [approved | preliminary]

Type Recursive Drawing =
[DwgStatus: Dwg-Status; ParentDwg: Drawing; Sublist: ListOf Drawing]

Type Job =
[JobDrawing: Drawing; NextMc: Machine; MaterlStatus: Boolean; Jobtype: Job-Type; Oper_Pending: Oper-Sequence; DueDate: Date; CurrentCost: real; Update (Job': Job); Function Job']

let x = Hd (Oper_Pending) in Cost = Cost + \* x.CostRate * x.Duration;

Oper_Pending = Tl (Oper_Pending);
NextMc = x.MachineEnd

Type Job-Packg = [ListOf Job]

Type Abstract Operation =
[MachineType: Machine-Kind; Machineld: Machine; Resource': Resource; Resource': Resource; TimeFrom: Time; TimeTo: Time; Duration: Time-interval]

Type Drilling = Is-A Operation +
[OperType: Boring/Finishing/Drilling; NC-Tape-No: int; CostRate: int]

Type Milling = Is-A Operation +
[OperType: Planning/Slotting; NC-Tape-No: int; CostRate: int]

Type Operation_Seqn = [ListOf Operation]

Type Abstract Machine =
[MachineStat: MachineStatus; Location: string; MaxJobSize: (int, int, int); MaxJobWt: int; MachineType: Machine-Kind]

Type Abstract MachineKind = [Drilling-MC | Milling-MC]

Type Drilling-MC = Is-A Machine +
[MaxDia: int; TableSize: (int, int); OperType: Drilling/Boring/Finishing]

Type Milling-MC = Is-A Machine +
[TableSize: (int, int); OperType: Planning/Slotting]

Type MachineStatus =
[State: busy/idle/breakdown/reserved; FromTime: Time; ToTime: Time]

Type Abstract Engine =
[Weight: int; Hp: int; RpmRange: (int, int); CoolingType: string]

Type Mar_Eng = Is-A Engine +
[ModelNos: int; ModelName: string]

Type Mar_EngBlock = Is-A Mar-Eng +
[JobSize: (int, int, int); JobWt: int]

Type Mar_Eng_Assly = Is-A Mar-Eng +
[JobSize: (int, int, int); JobWt: int]

Type Abstract JobType = [Mar_EngBlock | Mar_Eng_Assly | Auto_Eng_Assly | Auto_EngBlock]

Type Abstract MachineQueue =
[Machineld: Machine; Machinetype: Machine-Kind; WaitList: ListOf (Job, DueDate)]

Type DrillMcQueue = [ListOf (Mch: Machine, Mcq: MachineQueue)
where Mch.Machine_Kind = Drilling_Mc = Mcq.MachineType]

Type ShopScheduler = [schedule (Job': Job; Mc': Machine);
Function Job', Mc':
Job.Update (Job');
case
.Job'.NextMc = Drilling_Mc then DrillLoader.AssignQueue (Job', Job'.NextMc)
else MillLoader.AssignQueue. (Job', Job'.NextMc);
if Mc'.MachineType = Drill then DrillLoader.Load (Mc')
else MillLoader.Load (Mc')
Endfun]

Type DrillLoader =
[Joblist: DrillMcQueue; Load (Mc': Machine): Function Mc'.
let x = Hd (Joblist.Mc'.Mcq. Waitlist) and y=Mc'.MachineType and z=x.Hd (Oper_Pending) in
if Suitable (x, y, Mc') and Available (x, y, Mc'; CurrentTime) then
Joblist.Mc'.Mcq.Waitlist = Ti (Joblist.Mc'.Mcq.Waitlist);
Mc'.Status.State = Busy;
Mc'.Status.State.FromTime = CurrentTime;
Mc'.Status.State.ToTime = CurrentTime + z.Duration
Endfun];
AssignQueue (Job': Job; Mc': Machine); Function Job', Mc':
(Mc'.Mcq.Waitlist = Mc'.Mcq.Waitlist @ (Job', Job'.DueDate);
Mc'.Cell_Controller.Execute (Job')
Endfun]

Relation Suitable (x: Job, y: Operation, z: Machine) =
Smaller (x.JobType.JobSize, z.MaxJobSize) &
Lesser (x.MaterlStatus = True & y.Resource.State = Idle &
x.JobDrawing.DwgStatus = Approved.

Relation Available (x: Job, y: Operation, z: Machine, CurrentTime: Time) =
x.JobDrawing.DwgStatus = Approved.
Functions. Lambda expressions are Semlog’s primary mechanism for evaluating functions. A function can be defined using the lambda expression \( \lambda x.e \), or

$$\text{function } (x)(y).e,$$

where \( x \) is the parameter of the function whose type is \( y \), and \( e \) is the expression (given in terms of the parameter) that is used to compute the function’s value. The function type is given as in the language ML. Semlog functions are first-class values and can be passed as arguments to other functions or result from function evaluations. The system treats functions like any other object.

Expressions. Expressions in Semlog can contain variables, records, variants, function definitions, function applications, record fields, and case statements. Expression forms in the language include constants, identifiers, logic variables, fields, and case statements. An identifier is a symbolic name denoting a value. Logic variables have meaning only in the context of facts, rules, and queries in the language. Field selection is used to extract a component of a record and is denoted by \( r.f \), where \( r \) is the record and \( f \) is the field tag. The list operators \( \text{Hd}, \text{Tl}, \text{and @} \) obtain the head of a list, the tail of a list, and concatenate two lists, respectively.

Programming phases. Programming in Semlog has three phases:

1. **Specifying the domain.** The user enters a series of type (class) declarations describing the various entities of the domain, and relation declarations specifying associations between types.

2. **Creating the objects, facts, and rules for a particular instance of the domain.** The user creates domain objects by creating instances of entity types. Once created, a domain object can be bound to an identifier. Facts state associations among objects in the system, and rules express facts that depend on other facts. All the domain objects, facts, and rules entered in this phase must be type-consistent with the domain specification of Phase 1, ensured through type checking.

3. **Interacting with the system.** This phase is similar to stating queries or theorems to be proved in a Prolog environment.

**An example**

Let’s assume that a real-time Semlog control system is interfaced with a manufacturing system. Changes in the manufacturing system send messages to software objects in the control system via hardware interrupts. These software objects respond by sending instructions to other software objects and to hardware objects such as machine tools and conveyors. We use an engine-manufacturing shop to model such objects and relationships as assemblies, subassemblies, drawings, operation sequences, manufacturing schedules, machine classes, machines, interstage inventories, and job costs.

This example uses the following types (defined in the sidebar on page 60):

- ShopScheduler, DrillLoader
- MachineQueue, DrillMcQueue
- Job, Job Package
- Operation and its subtypes, Drilling and Milling
- Machine and its subtypes, Drilling_Mc and Milling_Mc
- Engine and its subtypes, Marine_Eng and Auto_Eng, and their subtypes, Mar_Eng_Axsl and Mar_EngBlock
- Drawing
- Dwg_Status

**Our approach aims to reduce the total life-cycle cost of developing the large and complex software systems required for FMS control.**

To effectively model the information and decision processes in FMS control, we must describe the nature of individual objects, the interrelationships among domain objects, the classification of objects, and the nature of changes that these objects undergo. This is due to three main factors. First, complex objects are the natural way to describe this complex domain. Second, information about the domain is usually incomplete and becomes available only incrementally. Third, the underlying database should take a more active role in deducing relationships rather than being a passive repository of data.
Current knowledge representation technologies do not provide all the primitives necessary for developing FMS control software. They do allow for different aspects to be modeled concisely, and thus are suitable for different classes of problems. Using primitives, strong typing, functional expressiveness, and precise semantics, our modeling environment describes objects naturally and succinctly, without recourse to obscure data structures. Our approach aims to reduce the total life-cycle cost of developing the large and complex software systems required for FMS control. By making it possible to program a large and complex control system with many interacting objects, this approach leads to improved software robustness and correctness, high software productivity, and easy maintainability.

Modeling the whole FMS environment in a realistic manner requires the use of new approaches. Our primary goal in this project was to lay the foundation for the methodological basis to design the control software for an FMS. We do not have working applications of our results; to develop such applications, we plan to study the large-scale modeling of existing FMS environments using Semlog.

Acknowledgments

We thank the anonymous referees and the participants of the 1991 Nashville joint TIMS/ORSA meeting for their comments. We also acknowledge financial support from the University of Texas at Austin and the University of Massachusetts at Boston.

References


Abhijit Chaudhury is assistant professor of management and information science at the University of Massachusetts, Boston. He received his PhD in MIS from Purdue University. His current research interests include artificial intelligence and information economics and their applications in MIS.

Chaudhury can be reached at the Dept. of Management Science and Information Systems, College of Management, University of Massachusetts, Boston, MA 02125.

Sukumar Rathnam is a doctoral candidate in information systems at the University of Texas at Austin. He holds a BTech in computer science from the Indian Institute of Technology, Madras, and an MBA from the Indian Institute of Management, Ahmedabad. His research focuses on computer graphics, object-oriented programming, programming language theory, and coordination theory.

Rathnam can be reached at the Dept. of Management Science and Information Systems, CBA 5.302, Univ. of Texas at Austin, Austin, TX 78712-1175, (512) 474 2771, or e-mail at sukumar@emx.utexas.edu