Knowledge-Based Automation of a Design Method for Concurrent Systems

Kevin L. Mills, *Senior Member*, *IEEE*, and Hassan Gomaa, *Member*, *IEEE*

Abstract—This paper describes a knowledge-based approach to automate a software design method for concurrent systems. The approach uses multiple paradigms to represent knowledge embedded in the design method. Semantic data modeling provides the means to represent concepts from a behavioral modeling technique, called Concurrent Object-Based Real-time Analysis (COBRA), which defines system behavior using data/control flow diagrams. Entity-Relationship modeling is used to represent a design metamodel based on a design method, called COncurrent Design Approach for Real-Time Systems (CODARTS), which represents concurrent designs as software architecture diagrams, task behavior specifications, and module specifications. Production rules provide the mechanism for codifying a set of CODARTS heuristics that can generate concurrent designs based on semantic concepts included in COBRA behavioral models and on entities and relationships included in CODARTS design metamodels. Together, the semantic data model, the entity-relationship model, and the production rules, when encoded using an expert-system shell, compose CODA, an automated designer's assistant. Other forms of automated reasoning, such as knowledge-based queries, can be used to check the correctness and completeness of generated designs with respect to properties defined in the CODARTS design metamodel. CODA is applied to generate 10 concurrent designs for four real-time problems. The paper reports the degree of automation achieved by CODA. The paper also evaluates the quality of generated designs by comparing the similarity between designs produced by CODA and human designs reported in the literature for the same problems. In addition, the paper compares CODA with four other approaches used to automate software design methods.

Index Terms—Automated software engineering, knowledge-based software engineering, software design, concurrent and real-time system design.

1 Introduction

Software engineering researchers and practitioners strive to improve the quality of software products by increasing the discipline used during software development. One means of increasing discipline entails the development and application of software design methods and supporting notations. A software design method provides a methodical, consistent, and teachable approach that defines what decisions a designer needs to make, when to make them, and, importantly, when to stop making decisions [1]. In addition, a software design method provides a consistent notation that can improve communication among those who must review and understand the meaning of a design. In effect, a software design method encodes knowledge about good design practices into a form that designers can use to construct software designs. For these reasons, numerous software design methods have been proposed and practiced [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. Some researchers attempt to enhance the utility of software design methods by providing automated support. This paper describes and evaluates one such approach to automating a software design method.

- *K.L. Mills is with the National Institute of Standards and Technology, Building 820, Mail Stop 8920, Gaithersburg, MD 20899. E-mail: kmills@nist.gov.*
- *H. Gomaa is with George Mason University, 4400 University Drive, Mail Stop 4A4, Fairfax, VA 22030-4444. E-mail: hgomaa@gmu.edu.*

Manuscript received 16 Nov. 1998; revised 6 Sept. 2000; accepted 21 Feb. 2001.
Recommended for acceptance by A.M.K. Cheng.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 108269.

Unlike previous approaches to design automation, the approach described and evaluated in this paper develops and exploits an underlying metamodel that can represent and reason about instances of requirements models, design models, and the relationships between the two. As a result, the approach described here can check instances of designs for consistency and completeness against the metamodel, can track traceability between the requirements model and the evolving design model, can automatically capture design rationale, can take different design decisions depending on whether interacting with a novice or experienced designer, and can vary the generated design to account for general design guidelines or to account for differences in target implementation environment.

After describing the motivation for design automation research in Section 2, the paper discusses some previous approaches to automate software design in Section 3. In Section 4, the paper briefly introduces CODARTS (COncurrent Design Approach for Real-Time Systems), a software design method for concurrent and real-time systems, and, then, proposes a knowledge-based approach to automate CODARTS in Section 5. The proposed approach leads directly to CODA (COncurrent Designer's Assistant), an automated designer's assistant. In a fragment from a case study, presented in Section 6, the paper describes the use of CODA to generate a design for an automobile cruise-control subsystem. Following the case study, the paper evaluates in Section 7 the performance of CODA when used to generate ten different concurrent designs for four real-time problems. In Section 8, the paper discusses the contributions of CODA, as compared with some previous approaches to automate software design methods. Section 9 considers future research.

2 Benefits from Automating Software Design Methods

Using automated support for software design methods can lead to several benefits. First, automation can improve the rigor with which a software design method is applied. Automation can ensure that a designer does not overlook any of the myriad details associated with the design process. Automation can establish that constraints levied on a design are satisfied, or that any unsatisfied constraints are brought to the designer's attention. Second, automation can improve a designer's ability to generate alternate designs. Since automation can speed up the generation of designs without sacrificing rigor, a designer can more readily produce several designs from one problem model. Third, automation can reduce the variability among the types of designs generated by various designers. Reduced variability of form can increase the ability of customers, analysts, and programmers to understand and compare designs. Fourth, automation can improve the performance of inexperienced designers both immediately, by making default decisions, and, gradually, by explaining default design decisions to the designer. The work described in this paper was motivated by the desire to produce an automated design assistant that would realize these benefits, while also advancing the state-of-the-art in automated design systems.

3 Previous Work on Automating Software Development

A number of researchers have proposed approaches for automating software development. The field exhibits a long history of attempts to automatically generate operational programs from requirements specifications. Much of that work failed to achieve the difficult goals envisioned. Realizing the difficulty of automated software generation, several researchers turned to the application of artificial intelligence techniques to provide automated assistance for the software design phases alone. Some researchers aimed at high-level design, while others focused on detailed design. Most of this work was disconnected from any particular software design methods that human designers applied and, so, the results have met with little success. Learning from these failures, some researchers have attempted to provide automated support for specific design methods with which human designers are already familiar. The work described in this paper can be classified in this latter category. The sections that follow provide a brief review of related research, discussed in three categories: 1) automatic programming, 2) automating software design, and 3) automating software design methods.

3.1 Automatic Programming

Unlike automated, design assistants, which help a human analyst complete a single, if essential, transformation in the software development process, automatic-programming systems [34], [35], [36], [37], [38] attempt to perform, without human intervention, every transformation required to generate a working implementation from an initial specification of user requirements. A different form of automatic programming, end-user programming, enables a computer-naive user to interact with an intelligent agent to select, exercise, evaluate, and modify an application program. End-user programming [39] requires no formal specification of requirements; in fact, the user need only bring the ideas in his head to a computer terminal to begin the process.

Numerous problems block success with automatic programming. Many automatic programming research projects seem to be limited to a single, small domain. Even in such projects the number and type of transformations required to convert a moderate specification into a program can be enormous. In addition, the automatic generation process produces a huge repository of data that can be difficult to manage. Further, the knowledge contained in an automatic programming system is dispersed widely and, thus, modifying such a system can be challenging. When an automatic programming system produces incorrect results, end users tend to examine the target code for the cause of the errors. Such an approach to software debugging, reminiscent of programmers who would modify the object code produced by a faulty compiler, can be costly, risky, and unproductive. Experience to date indicates that automatic programming will remain confined, for the foreseeable future, to single, small application domains.

The end-user variant of automatic programming systems overcomes the limits of a single, small domain, but at a cost. End-user programming systems require that a user sort through a range of problem-solving strategies in an effort to determine which approach might best meet his problem. After selecting an approach, the user must interact with the program over a long period of time until the performance of the program meets the user's expectations. As one possible outcome of this prolonged interaction, the user or the system might realize that the initial problem-solving strategy was, in fact, wrong. The basic approach to end-user programming seems to be educated guess, followed by trial and error refinement. Few users have the patience for such an approach to programming.

3.2 Automating Software Design

Designing software requires that a designer possess both creativity and a capacity for complexity and detail. A number of researchers investigate automated approaches to assist designers with the intricacies of software design, without unduly restricting the creative aspects of the design process. Some researchers address design at the architectural level, while others consider assistance for detailed design. For example, Fickas' Critter [40], based on an earlier tool known as Glitter [41], provides an automated assistant that attempts to bridge between requirements and design for composite systems, those containing a mixture of human, hardware, and software components. Critter uses an artificial intelligence paradigm of state-based search, relying on a human user to provide the domain knowledge necessary to guide the search. Critter encapsulates only domain-independent, design knowledge. Critter and a human designer interact to develop a design to solve a domain-specific problem. To date, the results with Critter do not appear encouraging. Critter's limited reasoning techniques prevent its use on large software engineering problems; the analysis algorithms used in Critter prove too

slow for an interactive design system; Critter's knowledge-base and representation omit several classes of system design concepts.

Progress to date on automating detailed design does not look any better. A number of other researchers investigated methods of providing automated assistance for detailed design. Most such methods assume the existence of one or more architectural designs. The assistance then focuses on locating and modifying, or on creating, components that can be fitted into one of the existing architectures. In most cases, detail-design assistants [42], [43], [44], [45], [46] operate in a narrow domain. Automated assistants of this type might be useful once a designer has already developed a system architecture.

3.3 Automating Software Design Methods

Some researchers attempt to provide automated support for familiar and well-established design methods. Four previous approaches are described here. Three of the four approaches produce a sequential design, represented as structure charts, from a behavioral model represented by data flow diagrams (DFDs). The fourth approach produces a concurrent design that to be mapped directly onto a design simulator. Each of these approaches is described below, followed by a brief discussion of advances made by the research presented in this paper.

3.3.1 Cluster Analysis

A system called Computer-Aided Process Organization, or CAPO, embodies one approach to transform a data flow diagram into a structured design [16]. CAPO strives to free a designer from using structured design techniques, such as transform and transaction analysis, to create structure charts. CAPO represents a data flow diagram as a flow graph and then converts that flow graph into six matrices, used to compute an interdependency weight for the links joining each pair of transformations. Based upon the computed weights, CAPO converts the flow graph into a weighted, directed graph, and then uses a number of cluster analysis techniques to decompose that directed graph into a set of nonoverlapping subgraphs.

CAPO provides no automated traceability between the flow graph and the resulting structure charts; such mapping must be determined by human inspection. CAPO also provides no automated assistance for checking the completeness and consistency of the proposed structure charts. In addition, CAPO does not capture the design rationale used to propose the various structure charts. In fact, wide variations in proposed structure charts can be obtained without changing the structure of the flow graphs by manipulating various numbers assigned to elements of the input flow graph. CAPO generates alternate designs by using various clustering algorithms but does not consider aspects of the target environment that might suggest alternate designs.

3.3.2 Specification-Transformation Expert System

Tsai and Ridge [17] describe a Specification-Transformation Expert System (STES) that automatically translates a specification model (expressed as data flow diagrams) into a sequential design (expressed as structure charts). The STES, implemented using the OPS5 expert-system shell, encapsulates the Structured Design method of Yourdon and Constantine [15] in expert-system rules. STES represents both data flow diagrams and structure charts as structured facts. STES uses several textbook heuristics, including coupling, cohesion, fan-in, and fan-out, to guide the design process. Each data flow in a data flow diagram has an associated data dictionary entry that can be used by STES to gauge the degree of coupling between modules in a structure chart. An expert system has difficulty determining cohesion among functions and, so, STES consults a user for information required to make inferences about functional cohesion. STES attempts to maximize fan-in and tries to achieve a moderate span of control.

STES operates as a sequential set of phases. First, STES factors the data flow diagram into afferent, efferent, and transform-centered branches. This factoring results in a top-level design for the structure chart. Second, STES refines each module at the next level of the structure chart using textbook guidelines for coupling, cohesion, fan-in, and fan-out. Third, STES renders the resulting, multilevel, structure chart using a CASE system from Cadre Technologies.

The approach embodied in STES limits its application to small designs, amenable to the sequential processing paradigm known as "inputs-processing-outputs." In addition, the STES provides no automated checking for completeness and consistency of the generated structure chart. Traceability between the data flow diagram and the structure chart must be verified manually. STES does not capture the rationale for design decisions. Though consulting the designer at various times, STES does not temper the nature of such consultation based on the designer's level of experience. STES cannot generate alternate designs without changing the data flow diagram.

3.3.3 Formal Rule Rewriting

Boloix et al. [18] describe another approach, based on an entity-aggregate-relationship-attribute (EARA) model, to automatically transform data flow diagrams to structure charts. Here, transformation rules, based on set theory, convert data flow diagrams, described formally at the lowest level of decomposition using an EARA model, into a formal description of structure charts. A human analyst then improves the resulting structure charts.

The EARA approach provides no automated completeness and consistency checking for the generated structure charts. In addition, the approach fails to capture the rationale used to generate the structure charts. Nor does the approach give consideration to generating alternate designs based upon variations in the intended runtime environment for the system under design. When consulting the designer at numerous points in the design-generation process, the EARA method does not vary the scope and nature of this elicitation based on the designer's level of experience.

3.3.4 SARA Design Apprentice

Another approach, reported in the literature by Lor and Berry [19], transforms requirements into a design, but without using structure charts as the target. This semi-automated, knowledge-based approach, developed by Lor

as the subject of a PhD dissertation in the context of the System ARchitects Apprentice (SARA), a joint development of researchers at UCLA and the University of Wisconsin [47], builds on the SARA environment by providing automated assistance to help a designer transform a requirements specification into a SARA structural model and graph model of behavior, or GMB. Lor uses data flow diagrams and system verification diagrams to specify requirements. System verification diagrams provide a stimulus-response model of behavior that Lor uses to specify interactions among subsystems in a design. Lor uses data flow diagrams mainly to specify the interior of subsystems.

Lor chose a rule-based approach for his design assistant for two reasons. First, since the current set of rules for transforming requirements into SARA designs remains incomplete, locking the knowledge into a procedural program appears premature. Second, the sequence of rule firings provides a natural explanation facility for design choices. The design assistant encompasses 21 rules for building the structural model, 59 for synthesizing the control domain, and 37 for modeling the data domain. Lor's approach synthesizes a SARA structural model through a direct translation of the hierarchy of data flow diagrams; at the lowest level of decomposition, the data flows map to SARA domain primitives. Lor's approach also creates a SARA GMB from the stimulus-response model provided by the system verification diagrams, as well as from the data flow diagrams.

Lor reports that his research provides a better understanding of, and a methodical approach to, designing systems in the SARA environment. The rules encapsulated in the design assistant can be called syntactically complete because every requirements construct is covered. The rules cannot, however, be called semantically complete; alternative designs cannot be considered and the rules cannot always map each requirements element to the most concise design construct. A human designer must answer queries as the design progresses (to provide needed information and to indicate preferences) and must improve the generated design. Given the same requirements specification technique (i.e., system verification diagrams and data flow diagrams), Lor asserts that his approach could be adapted to other design representations by rewriting the rule consequents; however, since the most crucial step in Lor's approach entails developing formal definitions, represented by SARA design constructs, for every construct in his requirements language, adapting to another design representation would require that this most crucial step be repeated.

3.3.5 Advances Over Previous Approaches

The work described in this paper provides several advances over the previous, related research. First, the current work provides an underlying metamodel that describes components, relationships, and constraints that designs must satisfy. This allows automated checking of design instances for completeness and consistency with respect to the metamodel. Such checking enables errors that can easily be made by human designers to be uncovered. Of course, the automated design generator included in the approach generates designs that should readily pass the completeness and consistency checks. Second, the current work provides automatic capture of design rationale. This allows a human designer to understand how a design decision was made. Such rationale can be used when an experienced designer changes a design, or when an inexperienced designer is learning the design method. Third, the current work provides two modes of operation: experienced and inexperienced. In experienced mode, the design generator will elicit information and assistance from the designer, as needed to address various subtleties in an evolving design, or to seek additional information that can help to resolve ambiguities or to provide a better design. In the inexperienced mode, the design generator uses default assumptions to address subtleties, to resolve ambiguities, and to make decisions about design optimizations. Finally, the current work enables the generation of designs that depend on characteristics of the target hardware and operating system. Using this feature, a designer can generate design variations more suited to particular target environments. A more detailed comparison between the current work and these previous approaches is given in Section 8.

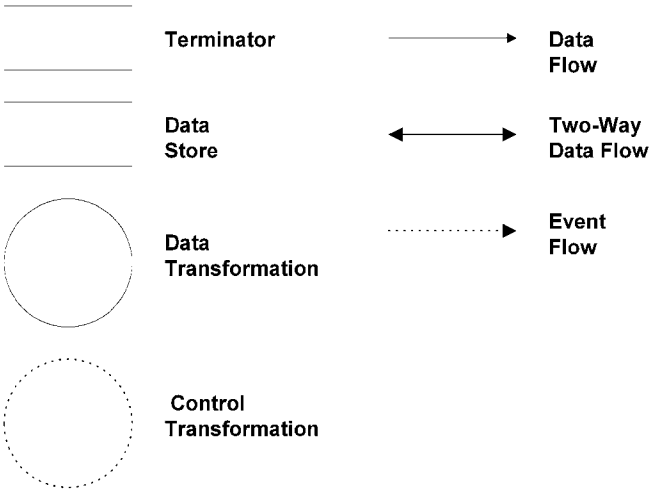


Fig. 1. Syntactic elements for composing COBRA data/control flow diagrams.

4 CODARTS: A Software Design Method for Real-Time Systems

CODARTS, or COncurrent Design Approach for Real-Time Systems, is a software design method for concurrent and real-time systems. CODARTS [7] uses criteria for information hiding and task structuring to form a concurrent design, including both tasks and information-hiding modules, [20] from a behavioral specification. CODARTS begins by using COBRA (Concurrent Object-Based Real-time Analysis) to analyze and model a system under design. COBRA uses RTSA (Real-Time Structured Analysis) notation, as summarized in Fig. 1. However, COBRA provides an alternative to the RTSA [8], [14] decomposition strategy that includes guidelines for developing an environmental model based on the system context diagram and defines structuring criteria for decomposing a system into subsystems and for determining objects and functions in each subsystem. Finally, COBRA includes a behavioral approach, based on event sequencing scenarios, for determining how the objects and

functions within a subsystem interact. A COBRA specification is documented as a hierarchical data/control flow diagram (D/CFD) and a data dictionary. A D/CFD has a state-transition diagram for each control transformation and a minispecification for each data transformation. Fig. 15 shows a fragment of a COBRA D/CFD for an automobile cruise-control application.

Once a COBRA specification exists, CODARTS provides four steps for generating a concurrent design:

1. task structuring,
2. task interface definition,
3. module structuring, and
4. task and module integration.

First, CODARTS task structuring criteria assist a designer in examining a COBRA specification to identify concurrent tasks. The task structuring criteria, consisting of a set of heuristics derived from experience obtained in the design of concurrent systems, can be grouped into four categories: input/output task structuring criteria, internal task structuring criteria, task cohesion criteria, and task priority criteria. In a given design, a task may exhibit several criteria and many tasks may exhibit the same criteria.

The input/output and internal task structuring criteria help to identify tasks based upon how and when a task is activated: periodically, based on the need to poll a device or to perform a calculation, or, asynchronously, based on an external device interrupt or on an internal event. The task cohesion criteria helps a designer to identify COBRA objects and functions that can be combined together in the same task. Single tasks might be formed wherever a set of transformations must be performed sequentially (sequential cohesion). When a set of tasks can be executed with the same period or with a harmonic period, those tasks might also be combined (temporal cohesion). When a set of transformations performs closely related functions, those transformations might be included in the same task (functional cohesion). The task priority criteria prevents a designer from combining tasks that might need to execute at substantially differing priorities.

As a second step, CODARTS provides guidelines for defining interfaces between tasks. Once tasks are defined, data and event flows from a COBRA specification can be mapped to intertask signals or to tightly or loosely coupled messages, depending on the synchronization requirements between specific pairs of tasks.

As a third step, CODARTS includes criteria, based on information hiding, to help a designer identify modules from the objects and functions in a COBRA specification. In general, the CODARTS module structuring criteria forms modules to hide the details of device characteristics, data structures, state-transition diagrams, and algorithms.

Finally, once both the task and module views of a concurrent design exist, CODARTS provides guidelines to help a designer combine the independent views into a single, consistent design. Each task represents a separate thread of control, activated by some event: an interrupt, a timer, an internal signal, or a message arrival. Each module provides operations that can be accessed by the tasks in a design. CODARTS helps a designer establish the control flow from events to tasks and then on to operations within modules.

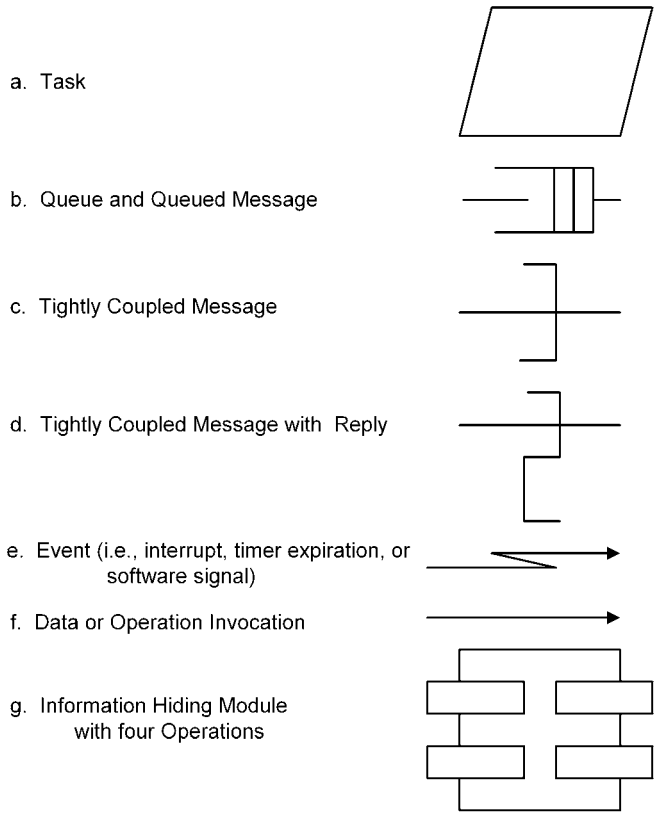


Fig. 2. Some key icons from the CODARTS graphical notation.

The results of applying CODARTS are documented in the form of a software architecture diagram and an accompanying set of task and module specifications. Some of the key icons in the graphical notation are illustrated in Fig. 2. The contents of the task and module specifications are discussed elsewhere [7]. Fig. 16 gives an example of a CODARTS design corresponding to the COBRA requirements model shown in Fig. 15.

5 Automating CODARTS

CODARTS provides design-structuring criteria to help a designer in structuring a software system into components. These criteria, expressed as heuristics or guidelines, are based on real-world experience in designing concurrent and real-time systems and have evolved over several refinements of the design method [30], [48], [49], [7]. Furthermore, the criteria have been validated through widespread use on industrial projects [50]. As the CODARTS structuring criteria are aimed at a human designer, they are described textually in considerable detail with the aid of examples [7]. A key challenge for automating CODARTS was to codify these natural language heuristics as production rules that could be processed by a machine, capturing all the different cases, and subtleties, addressed by the heuristics. Once codified, and encoded using an expert system shell, this design knowledge forms the basis for CODA, an automated assistant for designers of concurrent and real-time systems. The following discussion explains the ideas underlying CODA.

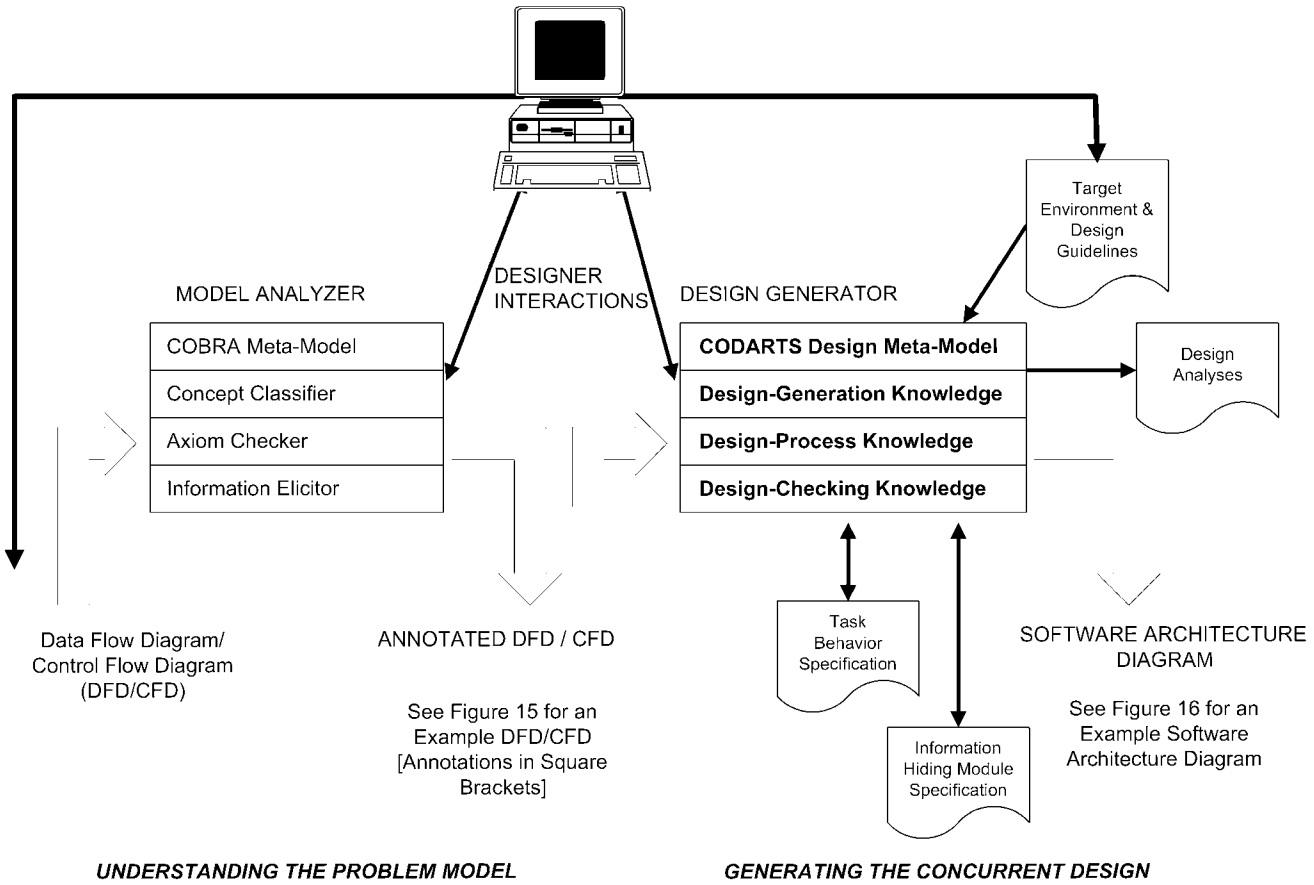


Fig. 3. Conceptual architecture for concurrent designer's assistant, CODA.

5.1 Overview of CODA

Fig. 3 illustrates one view of the architecture for CODA. Given a data/control flow diagram and a description of the intended target environment, along with any design guidelines, CODA largely automates the process of generating a concurrent design. The resulting design consists of a software architecture diagram, initial specifications for tasks and for information hiding modules, and consistency and completeness analyses of the generated design. Conceptually, CODA consists of two main components: a model analyzer and a design generator. The model analyzer converts a syntactically described flow diagram into a flow diagram annotated with semantic concepts from COBRA (see Fig. 15 for an example of an annotated flow diagram). The model analyzer consists of four knowledge bases:

1. an analysis metamodel that describes relationships among semantic concepts within a specific analysis method,
2. concept classification rules that perform inferences on instances of semantic concepts within the analysis metamodel,
3. axioms that define relationships required and prohibited among semantic concepts in the analysis metamodel, and
4. information elicitation rules that can be used to obtain information not readily available from visual representations of the analysis metamodel.

For CODA to support a specific analysis method, these four knowledge bases must be created. In the work discussed in this paper, knowledge bases were built to support Concurrent Object-Based Real-time Analysis, or COBRA [7]. The model analyzer, discussed in detail elsewhere [21], [22], is described briefly in Section 5.2.

The design generator uses design knowledge from CODARTS to transform an annotated flow diagram into a concurrent design. The current paper focuses on the design generator, highlighted in Fig. 3, which consists of a design metamodel that encodes the entities, attributes, and relationships available to construct instances of CODARTS designs, and three knowledge bases that encode CODARTS design heuristics, process constraints, and consistency and completeness constraints, respectively. Section 5.3 describes the CODARTS design metamodel, the related consistency and completeness constraints, and the characteristics of target environments, as seen by CODA. Section 5.4 explains how CODARTS heuristics can be represented as rule sets. Section 5.5 discusses how constraints from the CODARTS metamodel can be represented as predicates, and how those predicates can be encoded as object-oriented queries that can be applied to instances of CODARTS designs. Section 5.6 gives a brief explanation of the techniques used to capture and access design rationale.

5.2 The CODA Model Analyzer

The starting point for CODA consists of a data/control flow diagram represented using the syntactic elements of

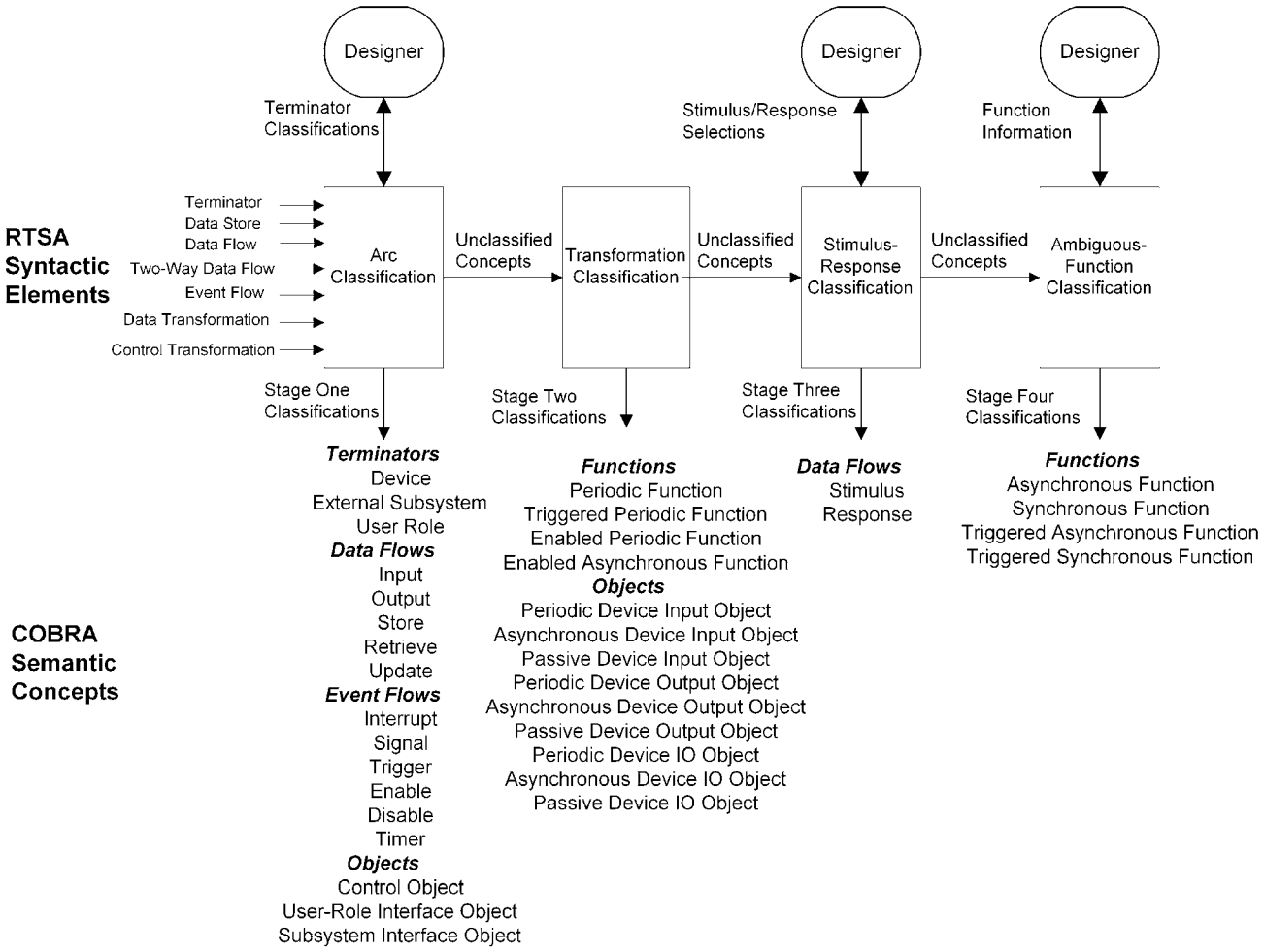


Fig. 4. An automated classifier for COBRA semantic concepts.

RTSA, as illustrated in Fig. 1. Before the CODA design generator can apply CODARTS heuristics, the model analyzer, working together with the designer where necessary, must classify the syntactical elements from RTSA flow diagrams as semantic concepts in COBRA. To accomplish this task, the model analyzer depends upon a COBRA metamodel, a concept classifier and axiom checker, and an information elicitor.

The COBRA metamodel, described elsewhere [21], comprises a taxonomy of semantic concepts [23], [24]. Each concept in the taxonomy can be constrained by a set of axioms [25]. Each leaf-level concept in the taxonomy must satisfy all axioms defined for the intermediate concepts along all its inheritance paths to the top of the taxonomy.

The actual classification of concepts on the flow diagram occurs through use of an automated concept classifier [26]. The concept classifier developed for CODA consists of a four-stage inference network [27], illustrated in Fig. 4. The classifier examines RTSA syntactic elements and classifies each as a concept in the COBRA taxonomy. Fig. 4 identifies the 36 leaf-level concepts in the COBRA taxonomy. Where ambiguity exists during classification, the concept classifier consults the designer. Where the designer cannot resolve the ambiguity, the concept classifier makes default decisions that have been encoded in the classification rules as the most likely outcome in the particular situation. To verify the work of the concept classifier, an axiom checker can ensure that every RTSA element is properly classified as one of the 36 COBRA semantic concepts shown in Fig. 4, and can ensure that each concept satisfies all required axioms.

The final component of the model analyzer elicits information from the designer, where such information cannot be derived directly from a flow diagram. In addition, newly classified concepts might require additional information in order to make subsequent design decisions. For example, if a control flow is classified as a timer, a positive period must be supplied for the timer. The information elicitor automatically identifies when additional information is necessary, prompts the designer for the information, and performs consistency checks on the information supplied. Fig. 15 illustrates the output of the COBRA model analyzer for a fragment of a D/CFD for an automobile cruise-control system.

5.3 The CODARTS Design Metamodel

The CODARTS design metamodel provides a basis for describing concurrent designs and for reasoning about those designs using automated methods. The design metamodel also provides for traceability between concurrent designs and elements of the data/control flow diagrams from which the design is generated. In addition,

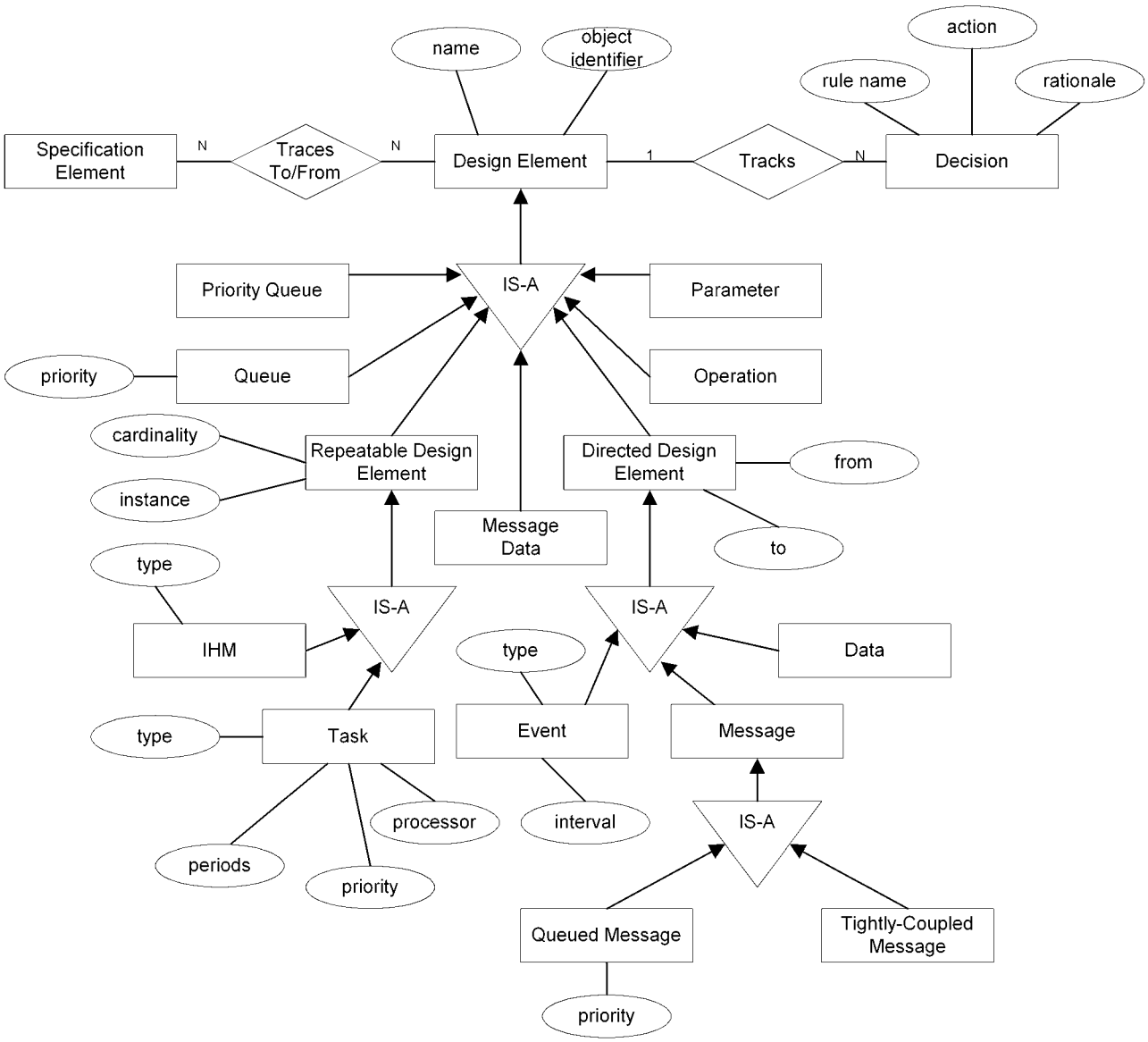


Fig. 5. E-R model of design entities composing the CODARTS design metamodel.

the design metamodel allows design decisions and associated rationale to be captured and organized automatically.

The CODARTS design metamodel consists of entities, attributes, relationships, and constraints. The entities, attributes, and relationships can be visualized conveniently using an Entity-Relationship (ER) diagram. The constraints cannot be shown in a visually appealing form. One class of constraints restricts the possible mappings between elements from a COBRA behavioral model and elements in a corresponding concurrent design. The second class of constraints defines restrictions among relationships in a concurrent design. These constraints permit instances of a design to be checked for consistency and completeness.

Fig. 5 illustrates the entities and associated attributes that compose the metamodel for concurrent designs. The figure also shows some inheritance relationships among those entities and depicts two key relationships in which all design entities participate. Every entity in the design metamodel is a named design element that possesses a unique object identifier within a given design. Each design element can track every decision made about it; thus, CODA captures design rationale, including the name of the rule (see Section 5.4) that executed the decision and the specific actions taken to update the design. In addition, each design element must trace from one (or more) specification element in a COBRA metamodel; however, certain constraints, given in Table 1, restrict this relationship to those that make sense. The remaining entities in Fig. 5 depict the semantic elements used in CODARTS to describe concurrent designs. In general, CODARTS designs consist of three types of entities: 1) repeatable design elements, 2) directed design elements, and 3) auxiliary design elements. Repeatable design elements include the main structural elements of a concurrent design: tasks and information-hiding modules. Directed design elements link together the structural components of a design. For example, messages are sent between tasks. Two types of messages can be exchanged between tasks: 1) queued messages and 2) tightly coupled messages.

TABLE 1
Constraints on Traceability from COBRA Behavioral Models to Concurrent Designs

Design Element	Traces from COBRA Semantic Concept(s)
Task	Control or Data Transformation
Information Hiding Module	Data Store, Data Flow, Control or Data Transformation, Two-Way Data Flow
Queue or Priority Queue	Signal, Stimulus, Control or Data Transformation
Message or Message Data	Control Event Flow, Internal Data Flow, Signal
Event	Control Event Flow, Normally-Named Event Flow
Data	External Data Flow
Operation	Data-Store Data Flow, External Data Flow, Interrupt, Transformation, Update
Parameter	Control Event Flow, Data Store, External Data Flow, Internal Data Flow, Signal

The "Tracks" and "Traces" relationships depicted in Fig. 5 apply to every design element. The "Tracks" relationship enables a history of design decisions to be associated with each element in a design. Similarly, the "Traces" relationship enables each design element to be associated with the flow-diagram symbols from which the element is derived. Other semantic relationships between design elements are depicted using a separate E-R diagram, shown as Fig. 6. Entities with the same name on both Fig. 5 and Fig. 6 represent the same design element, so the two E-R diagrams can be understood as two different views of a more complex model. Each relationship in Fig. 6 should be understood to be bidirectional, including both the relationship as shown and its inverse. For the most part, the relationships shown in Fig. 6 can be read intuitively.

Consider the relationships between Task and Message, as depicted on Fig. 6. A Task can send and receive many messages and each message must be sent and received by one Task. Further, a message may include Message Data, which carries information between tasks. Messages may be of two types, a Queued Message, which can be sent without causing the sending Task to block, or a Tightly Coupled Message, which causes the sending Task to block until the receiving Task accepts the message. Some messages require

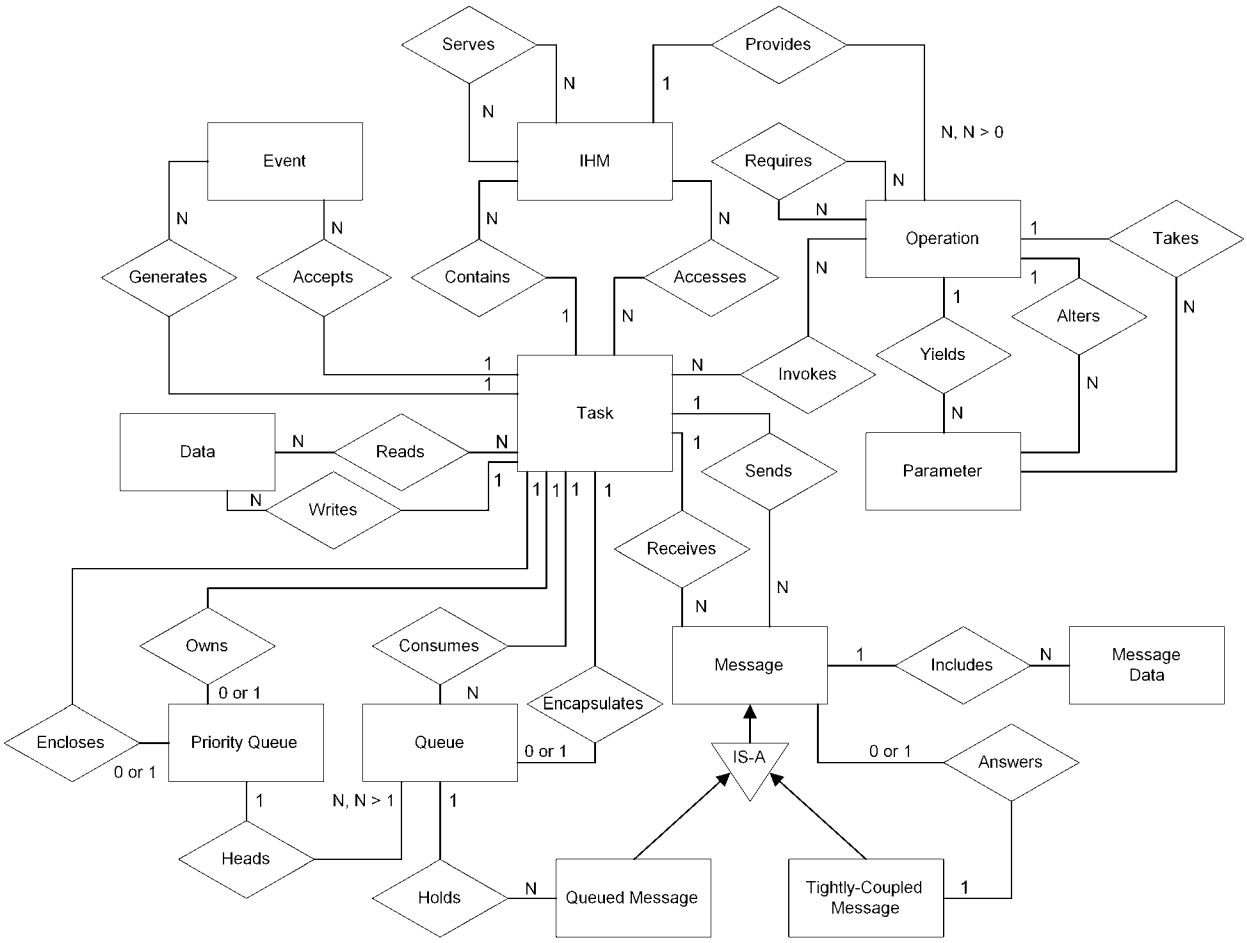


Fig. 6. E-R model of design relationships composing the CODARTS design metamodel.

a reply, as depicted by the Answers relationship. Note that a Tightly Coupled Message answers a message; thus, all replies cause the sending Task to block until the receiving Task has accepted the reply.

While Fig. 6 does depict cardinality constraints, more complex constraints do not appear on the E-R diagram. For example, each module in a given design is either contained in a task or is accessed by a task or another module. Such complex constraints are represented as predicates that must hold for valid instances of the design metamodel. These predicates, when expressed as knowledge-based queries, provide the design generator with the knowledge needed to check designs for consistency and completeness with respect to the design metamodel.

While many design decisions can be taken in the abstract, depending only on concepts represented in the design metamodel, other design decisions must account for specific characteristics of the target environment in which the design will execute. To account for such characteristics, CODA enables the designer to specify, for instance, the number of processors involved in a system, the type of interprocess communications mechanisms available, and the number of available task priority levels.

5.4 Design Generation Knowledge

Using the semantic concepts represented in the COBRA metamodel and the CODARTS design metamodel, along with some characteristics of the intended target environment, a human designer can apply various heuristics from the CODARTS design method to produce a concurrent design from a COBRA behavioral model. To automate design generation, heuristics from CODARTS must be formulated as expert-system rules [28] that can reason about data/control flow diagrams and evolving concurrent designs. The specific rules contained in CODA were developed from a natural language description of CODARTS design heuristics [7]. Since the requirements metamodel and the design metamodel were constructed from concepts contained in COBRA and CODARTS, the CODA rules can express CODARTS heuristics in terms relatively familiar to a human designer. Each rule consists of an if-then construction, where the antecedent matches a pattern of concepts in the annotated flow diagram, entities and relationships in the evolving design, or some combination. Many rules were proven to be very simple with only a single predicate in the antecedent. In some rules, a conjunction of as many as ten predicates was required to correctly specify the antecedent. Some complexity also arose associated with guiding the firing order when multiple rules might be satisfied simultaneously. To address these situations, a careful analysis of the CODARTS design process, as applied by human designers, identified which design criteria should take precedence over other criteria. This knowledge was encoded as six precedence levels. Each of the 126 design-generation rules was assigned one of the six levels. By comparing the designs produced by CODA against designs produced by human designers, as reported in the literature, the expert system rules were tested for validity. Where differences appeared between the designs generated by CODA and the designs reported in the literature, the reasons for the differences were identified and analyzed. Section 7.3 provides more detail related to this validation.

The expert system rules that formalized the CODARTS design heuristics were encoded as a partitioned repository of design-generation knowledge that the CODA design generator uses to transform flow diagrams into designs. Each knowledge partition corresponds to a step in the CODARTS design method:

1. task structuring,
2. task-interface definition,
3. information-hiding module structuring, and
4. task and module integration.

The execution of these four knowledge partitions must meet the process constraints imposed by the CODARTS design method. Task Structuring and Module Structuring are independent activities that must both be completed prior to integrating the task and module views. Task Structuring must be completed prior to defining the interfaces between tasks. A more detailed discussion follows for each knowledge partition.

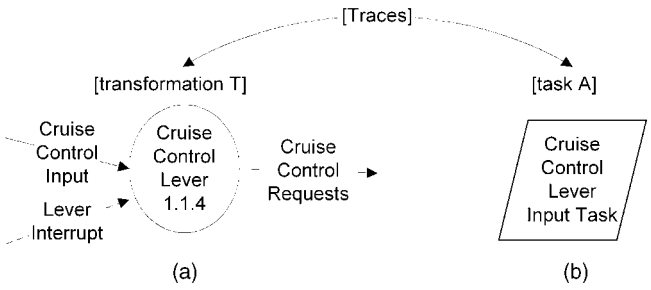
5.4.1 Task Structuring Knowledge

Task structuring knowledge, as encoded for use by the CODA design generator, consists of a sequence of four decision-making processes:

1. identify candidate tasks,
2. allocate remaining transformations to tasks,
3. consider task mergers, and
4. consider resource monitors.

Each of these processes consists of a set of production rules that search the flow diagram and the emerging design for matching patterns. When a matching pattern is found, the associated rule is activated, updating the emerging design according to actions specified in the rule. The first decision-making process, consisting of 11 rules, applies CODARTS heuristics to identify those transformations that can be allocated to input/output tasks and to internal tasks. The second process, encompassing nine rules, applies selected CODARTS cohesion criteria to allocate each of the remaining transformations to one or more of the tasks identified during the first process. The third process, comprising eight rules, examines the tentative task structure, applying additional CODARTS cohesion criteria to reduce the number of tasks by merging tasks where appropriate. The final process, requiring only two rules, identifies instances where a resource monitor task is needed to arbitrate access by multiple tasks to a single device. A few examples, taken from a case study presented in Section 5, will illustrate how CODARTS task structuring knowledge can be represented as production rules.

Fig. 7 shows how a transformation in a flow diagram model leads to the generation of an input/output task. Fig. 7a shows a transformation, Cruise Control Lever, activated by an interrupt. During preprocessing by the CODA model analyzer, Cruise Control Lever was classified as an "Asynchronous Device Input Object." Since such an object inherits the characteristics of an "Asynchronous Device Interface Object," Cruise Control Lever satisfies the antecedent of the rule shown in Fig. 7c. As a consequence of this rule, Fig. 7b shows that an asynchronous-device input task, [task A], is created and that a traceability link,




```
if
        transformation T is an Asynchronous Device Interface Object
then
        if      transformation T is an Asynchronous Device Input Object
        then    create an asynchronous device-input task A
        elseif  transformation T is an Asynchronous Device Output Object
        then    create an asynchronous device-ouput task A
        else    create an asynchronous device-input/ouput task A
        fi
        record the decision and rationale in the design history for task A
        denote the traceability between the transformation T  and task A
fi
```

(c)

Fig. 7. Heuristic for generating an input/output task.

[Traces], is established between the new task and the transformation. Not shown in Fig. 7, the decision and rationale are noted and added to the design history for the new task. Using the rule shown in Fig. 7c and two similar rules, the CODA design generator can identify all transformations that lead to CODARTS input/output tasks. Eight additional rules are required to apply CODARTS criteria for structuring internal tasks.

Fig. 8 describes one of the rules encoding the CODARTS criteria for identifying internal tasks. Fig. 8a shows a transformation, Maintain Speed, which was previously classified by the CODA model analyzer as an "Enabled Periodic Function." The rule defined in Fig. 8c matches transformations that are enabled and disabled by a control object and that execute periodically when enabled. The rule in Fig. 8c creates a task, shown as a design fragment in Fig. 8b, and links that task to the appropriate transformation from the flow diagram model.

Fig. 9 illustrates how two CODARTS criteria, functional cohesion and temporal cohesion, can be combined into a single rule that can merge tasks. When periodic tasks of identical type (functional cohesion) exhibit identical execution intervals (temporal cohesion) and each of those tasks represents a single instance, the tasks can be merged into one task. Fig. 9c specifies the rule that recognizes when tasks can be merged. Before the rule execution, Fig. 9a, the design consists of two periodic device-input tasks, each of which executes every 100 milliseconds. After the rule execution, Fig. 9b, these tasks have been merged to form a single task.

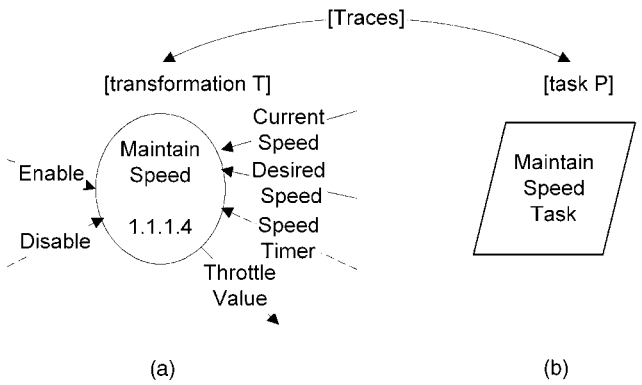
5.4.2 Task-Interface Definition Knowledge

After determining the tasks in a concurrent design, the CODA design generator can identify the subset of data and event flows exchanged among the tasks and can then map those flows to specific communication mechanisms between pairs of tasks. The CODARTS design method provides guidelines for selecting appropriate interface mechanisms. These guidelines can be represented as production rules encoded within five decision-making processes:

1. allocate external interfaces,
2. allocate control and event flows,
3. allocate data flows,
4. elicit message priorities, and
5. allocate queue interfaces.

The first process, consisting of five rules, creates the timer and interrupt events needed to activate particular tasks, maps data flows between tasks and devices into appropriate input and output data, and identifies the data and event flows exchanged between tasks. The second process, requiring eight rules, decides how to allocate event flows and control flows that are exchanged between tasks. Such flows can be allocated to software interrupts, to tightly coupled messages, or to queued messages. The third process uses 12 rules to map data flows onto either tightly coupled messages or queued messages. The fourth process has one rule that allows an experienced designer to indicate the relative priority of multiple messages arriving at specific tasks. The final process decides upon appropriate interfaces to receive queued messages. Such decisions depend upon the type of mechanisms available in the target environment and upon the priorities assigned when multiple queued messages arrive at a single task. Some examples will show how the CODARTS guidelines can be encoded as production rules.

Fig. 10 illustrates one rule for mapping an event flow onto a queued message exchanged between two tasks. The rule, defined in Fig. 10c, recognizes the case where an event flows from a device-input object in one task to a control

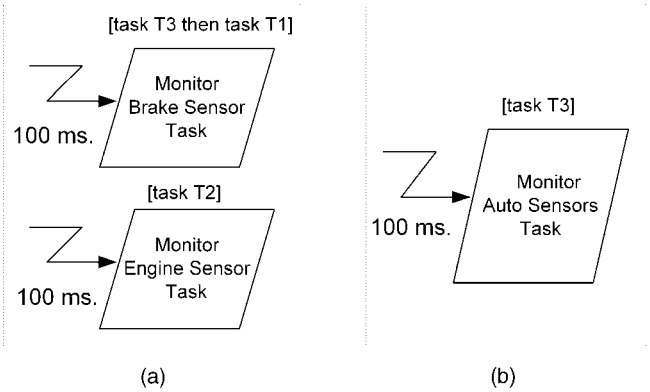



```
if
        transformation T is an Enabled Periodic Function
then
        create an enabled periodic task P
        record the decision and rationale in the design history for task P
        denote the traceability between transformation T and task P
fi
```

(c)

Fig. 8. Rule for generating an internal task.

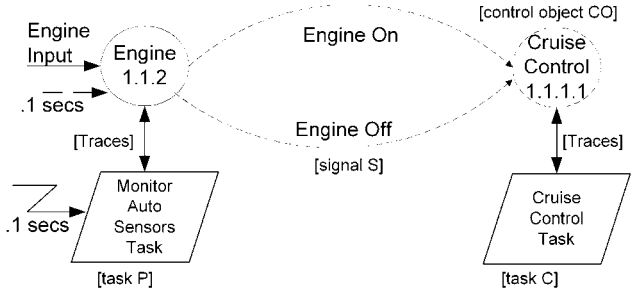
object in another task. In such cases, provided that the control object is not locked in one state awaiting the incoming event, the event flow can be mapped onto a queued message that is sent by the device-input task and received by the control task. One such case is shown in Fig. 10a, where a periodic device-input task, Monitor Auto Sensors, traces to a periodic device-input object, Engine, that sends an event flow, Engine Off, to a control object, Cruise Control, that traces to a control task of the same name. After the execution of the rule given in Fig. 10c, the CODA design generator updates the design fragment, as illustrated in Fig. 10b, to include a new queued message, Sensor Status Message, and two appropriate relationships, "Sends" and "Receives." In addition, the design generator records the traceability between Engine Off and the new Sensor Status Message, along with the decision and rationale. Once a message exists, additional event flows, such as Engine On in Figs. 10a and 10b, between the same set of tasks are allocated as event types in a parameter of the message.



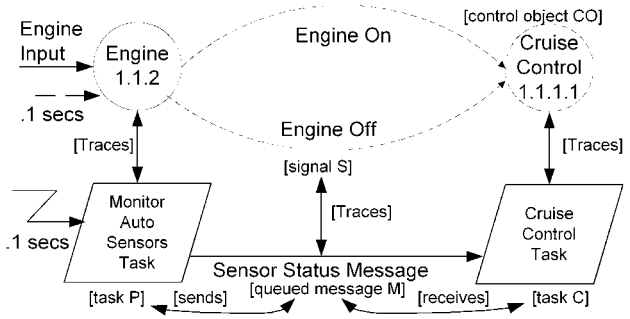

```
if
        task T1 is a periodic device-input task or a periodic device-output task
        or a periodic device-I/O task or a periodic internal task and
        task T2 is of identical type to task T1 and
        both tasks, T1 and T2, have a cardinality of one and
        tasks T1 and T2 have identical periods
then
        combine task T1 and task T2 into a single task T3
        record the decision and rationale in the design history for task T3
fi
```

(c)

Fig. 9. Rule applying functional and temporal cohesion to merge two tasks.



(a)



(b)

```
if
        task P sends a signal S to task C and
        the sink of signal S is a control object CO and
        control object CO is not blocked waiting for signal S and
        task P is a periodic or asynchronous device input task and
        task C is a control task and
        the source of signal S is a Periodic or Asynchronous Device Input Object
then
        create a queued message M from task P to task C
        establish the design relationship task P sends message M
        establish the design relationship task C receives message M
        denote the traceability between signal S and message M
        record the decision and rationale in the design history for message M
fi
```

(c)

Fig. 10. Rule to map event flow to a queued message.

Fig. 11a depicts a design fragment where one task, Cruise Control, receives queued messages from three sending tasks. In order to ensure that no incoming messages are lost and that each message is processed in an appropriate order, Cruise Control requires a queue to hold arriving messages. Fig. 11c defines one rule that generates a queue in a specific circumstance. When a task receives multiple queued messages at the same priority and the target environment provides message queues, a queue can be created to hold the incoming messages until the receiving task can consume them, as illustrated in Fig. 11b. An alternative mechanism can be selected using a different rule when the target environment does not provide message queues.

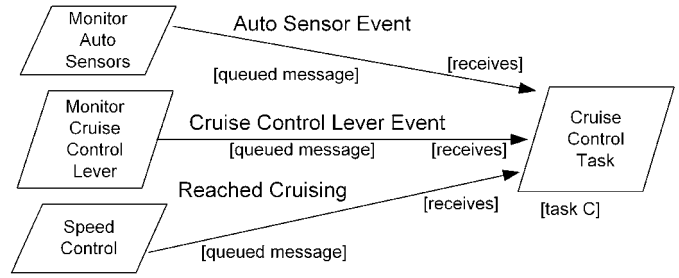
While many event flows and data flows can be mapped to queued messages, the CODARTS guidelines identify some situations requiring the use of tightly coupled messages. These guidelines can also be represented using production rules.

5.4.3 Information-Hiding Module Structuring Knowledge

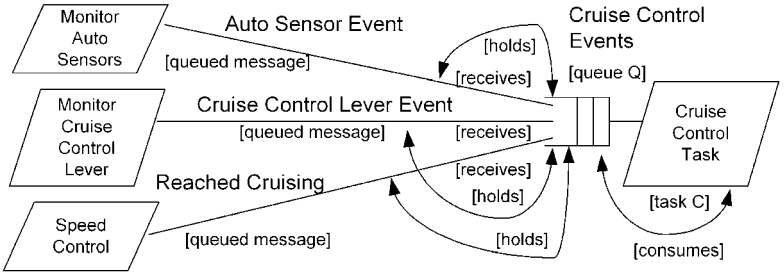
The CODARTS design method permits a designer to define information-hiding modules from a flow diagram model. This activity, called module structuring, can be carried out independently from the structuring of tasks. The CODA design generator provides support for module structuring by encoding CODARTS module-structuring guidelines into six decision-making processes:

1. identify candidate modules,
2. allocate functions to data-abstraction modules,
3. allocate remaining transformations to modules,
4. allocate isolated elements to modules,
5. consider combining modules, and
6. determine module operations.

The first process contains seven rules that identify nodes on a flow diagram that can be allocated to software modules. The second process consists of five rules that find specific transformations that can be mapped to functions provided by data-abstraction modules created during the first process. The third and fourth processes, each consisting of three rules, determine how to map the remaining transformations and data stores to software modules, either allocating each node to an existing module or to a new module. The fifth process, available only to experienced designers, contains one rule to identify situations in which a designer might prefer to combine software modules in the emerging design. Each such situation is referred to the



(a)



(b)

```
if
        task C receives queued messages at a single priority and
        message queues are available in the target environment
then
        create a queue Q for task C
        establish the design relationship task C consumes queue Q
        record the decision and rationale in the design history for queue Q
        for each queued message M to task C
                establish the design relationship queue Q holds message M
                record the descision and rationale in the design history for queue Q
        rof
fi
```

(c)

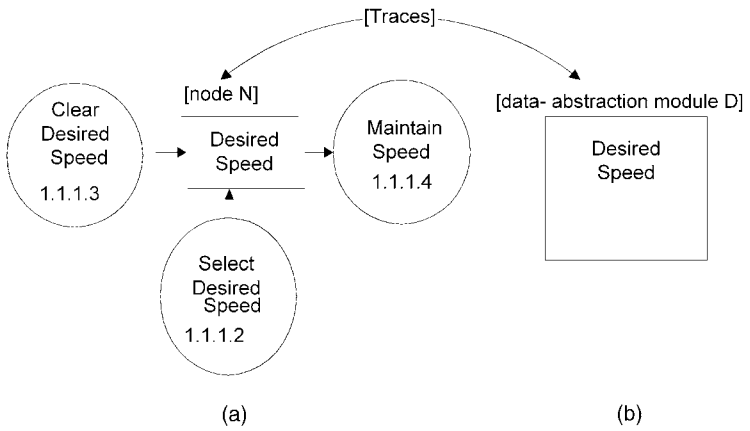
Fig. 11. Rule to allocate a message to a queue.

designer for a decision. The final process, requiring 21 rules, determines the specific operations provided by each module in the design, along with the parameters required by each operation. Two examples illustrate how CODARTS module-structuring guidelines can be represented as knowledge-based rules.

Fig. 12c gives a rule that encodes one of the CODARTS heuristics used to identify an information-hiding module from nodes in a flow diagram model. In this case, as shown in Fig. 12a, any data store, such as Desired Speed, which is accessed by multiple transformations, such as Clear Desired Speed, Maintain Speed, and Select Desired Speed, serves as the basis for a data-abstraction module. As a result of executing this rule on the flow diagram fragment shown in Fig. 12a, the CODA design generator produces the design fragment illustrated in Fig. 12b and links the data store to the new data-abstraction module. Other CODARTS heuristics lead to similar rules for identifying device-interface modules, user-interface modules, subsystem-interface modules, state-transition modules, function-driver modules, and algorithm-hiding modules. After the initial software modules are identified, another handful of rules can be used to allocate any remaining nodes to software modules.

After the final module structure is set, the CODA design generator, using 21 rules, can automatically construct the operations provided by each module and the parameters required for each operation. For example, Fig. 13c gives a rule for mapping functions to operations in three, specific types of modules: algorithm-hiding modules, function-driver modules, and data-abstraction modules. Figs. 13a and Fig. 13b show an example of the application of this rule to a design fragment. In this case, a data-abstraction module, Desired Speed, traces in part from a function, Select Desired Speed, which receives a control flow, Trigger, from a transformation, Cruise Control, which does not trace to the module named Desired Speed. Here, the function, Select Desired Speed, represents an operation that the data-abstraction module must provide to another module. Fig. 13b shows the updated design fragment created by an execution of the rule defined in Fig. 13c.

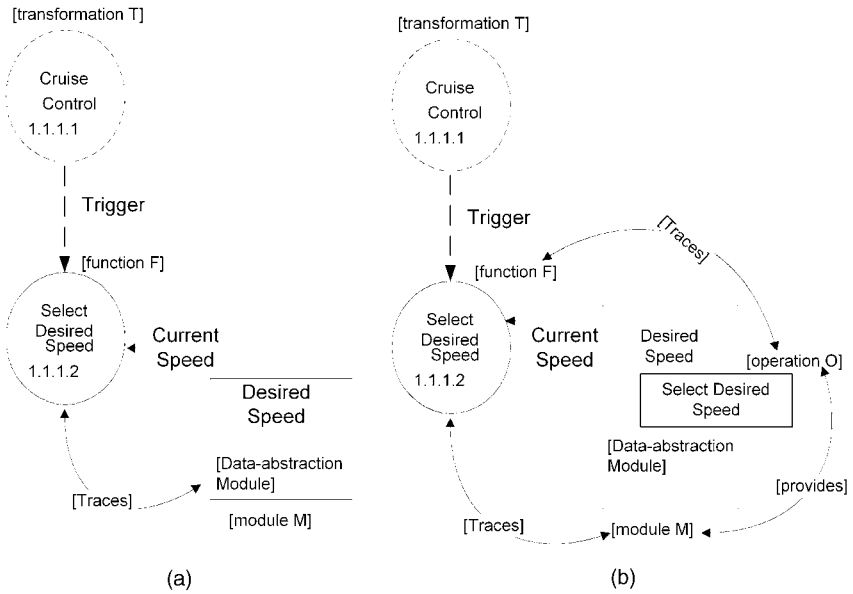
Twenty additional rules, similar in spirit to that shown as Fig. 13c above, complete the reasoning needed to generate all the operations and associated parameters required for all the software modules found in a design. After the design generator identifies both the modules and tasks in independent views, additional knowledge is needed to integrate the two views.



if node N is a data store and
node N is accessed by multiple transformations
then
create a data-abstraction module D
record the decision and rationale in the design history for module D
denote the traceability between node N and module D
fi

(c)

Fig. 12. Rule to identify a data-abstraction module.

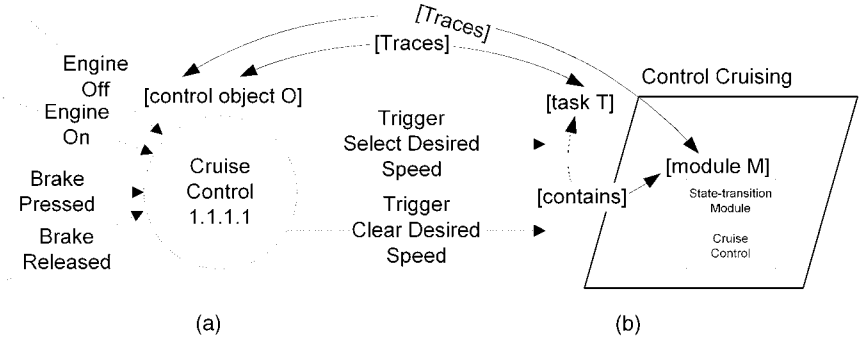


if module M is an algorithm-hiding module or a function-driver module or a
data-abstraction module and
function F traces to module M and
((function F recieves a trigger or enable or stimulus or signal from
transformation T and transformation T does not trace to module M) or
(function F does not receive any trigger or enable or stimulus or signal))
then
create an operation O with the same name as function F
denote the traceability between function F and operation O
establish the design relationship module M provides operation O
record the decision and rationale in the design history for module M

fi

(c)

Fig. 13. Rule to create a module operation.




```
if
        module M is a state-transition module and
        module M traces from a control object O and
        task T traces from the same control object O
then
        establish the design relationship that task T contains module M
        record the decision and rationale in the design history for module M
fi
```

(c)

Fig. 14. Rule to place a state-transition module in a task.

5.4.4 Task and Module Integration Knowledge

The CODARTS design method includes guidance about combining the task and module views into an integrated software design. The CODA design generator represents this guidance as production rules organized into three decision-making processes: 1) determine module placements, 2) link tasks and external modules, and 3) link external modules. The first process encompasses 10 rules that separate modules into two categories: those that are executed by a single thread and those that are executed by multiple threads. In addition, these rules place single-threaded modules within the task that executes them. The second process contains three rules that update multi-threaded modules to indicate which tasks invoke which operations in the modules. The third process contains four rules that identify which multithreaded modules require which operations in other modules. The following example illustrates the nature of task and module integration.

Fig. 14c gives a rule that identifies a single-threaded, state-transition module and places that module within the task that executes it. Fig. 14a shows a fragment from a flow diagram model, and Fig. 14b shows a fragment from a related design. The design generator defined the state-transition module, Cruise Control, during module structuring and the task, Control Cruising, during task structuring. Through the execution of the rule defined in Fig. 14c, the design generator links the two elements into an integrated view, specifying that the state-transition module is contained within the task. The "Contains" relationship denotes two properties: 1) the state-transition module is a single-threaded module executed only by the containing task and 2) the state-transition module is hidden lexically within the containing task.

5.5 Checking Completeness and Consistency

Each design produced by CODA is an instance of the design metamodel generated from an instance of the requirements metamodel. This fact enables CODA to check each design for completeness, relative to the input data/control flow diagram, and for consistency, relative to the constraints of the design metamodel. These checks can help a designer ensure that the design is well formed and that no details have been overlooked. CODA writes the results of this completeness and consistency check, along with an index of the tasks and modules generated for the design, as a design summary. In terms of completeness, CODA checks for the following conditions: 1) each transformation on the flow diagram is allocated to at least one task, 2) each transformation and data store is allocated to a module, and 3) each arc is allocated to appropriate elements in the design. During each check, CODA lists elements on the flow diagram that remain unallocated. For consistency, CODA checks that the design instance satisfies all constraints in the design metamodel, for example:

1. each module is either contained within or accessed by a task or another module,
2. each module provides at least one operation,
3. each operation is provided by a module,
4. each task receives at least one input and writes at least one output,
5. each internal event is both generated by and accepted by a task,
6. each external event and timer are accepted by a task,
7. each datum is either read or written by a task,
8. each message is either sent and received by a task or is carried as a parameter within another message, and
9. each queued message is held by a queue, and so on, through the 39 constraints the must be satisfied.

Each constraint violation is specifically reported so that the designer can investigate.

Completeness and consistency checking is implemented by treating each design and flow diagram instance as an object-oriented database. When checking for completeness, each predicate is encoded as an object-oriented query that

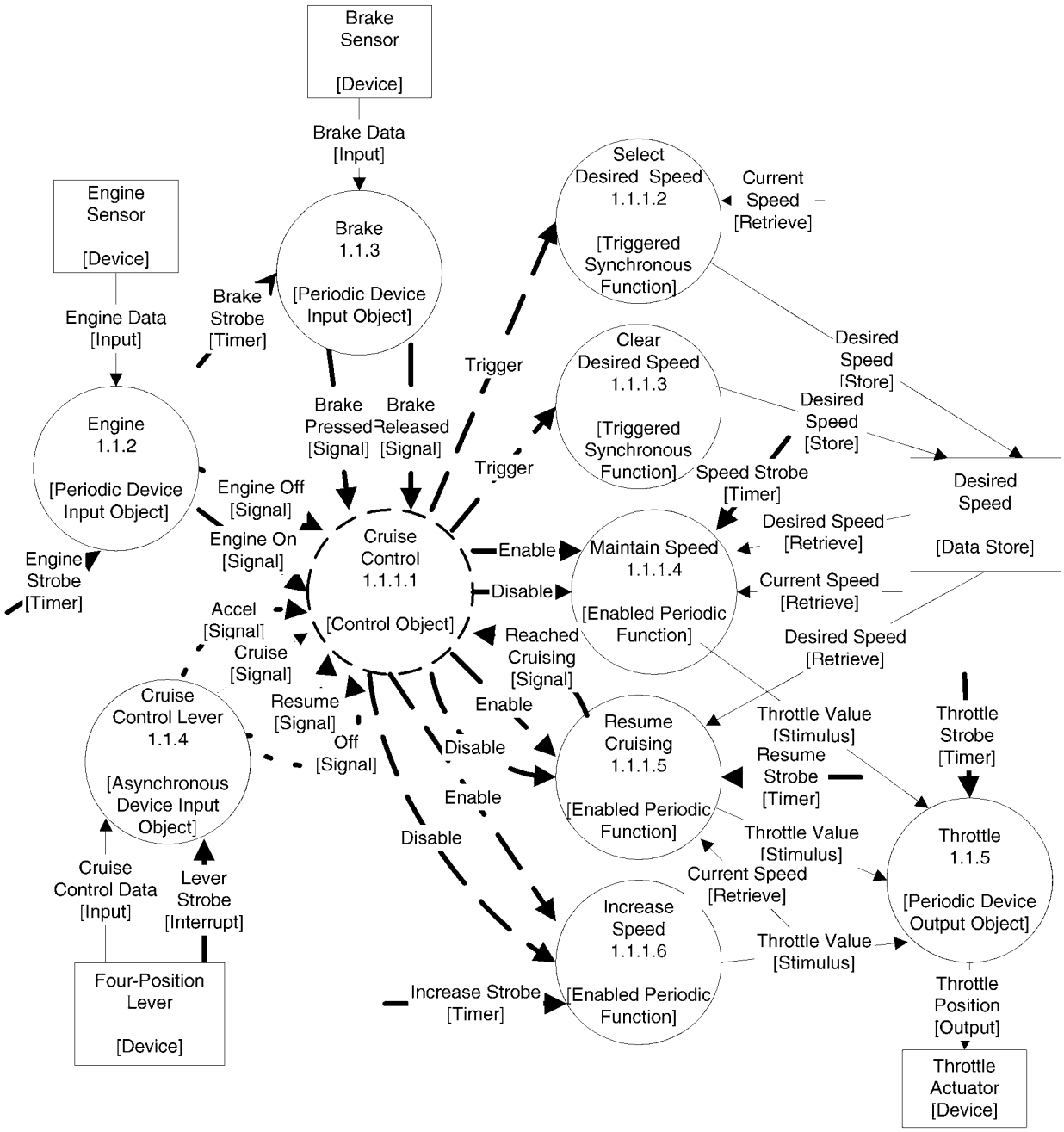


Fig. 15. Annotated Flow Diagram Model of the speed control aspects of an automobile cruise control subsystem (annotation shown in square brackets).

asks if a specific element on the flow diagram does not have traceability to an appropriate element in the design. When checking for consistency, each predicate, representing some constraint, is encoded as an object-oriented query against the database. Some of these queries can be quite complex. In most cases, a consistency query first checks for the existence of any single instance that violates a constraint. Once a specific constraint is violated, a query then checks for every instance that violates the constraint and reports each violation.

Viewing the metamodel as the schema for an object-oriented database opens up additional power because instances of metamodel can then be queried in an ad hoc manner. CODA implements some of these ad hoc queries in a canned form that a designer can use to answer a range of questions, such as which design elements follow from which elements on a flow diagram and vice versa. This approach also opens the door to recall the rationale used to generate each element on the design.

5.6 Capturing and Recalling Design Rationale

Whenever the CODA design generator takes a design decision the decision is recorded in human-readable text assigned to the relevant design element. Using the query interface provided by CODA, a designer can ask interactively for an explanation of any element in a design

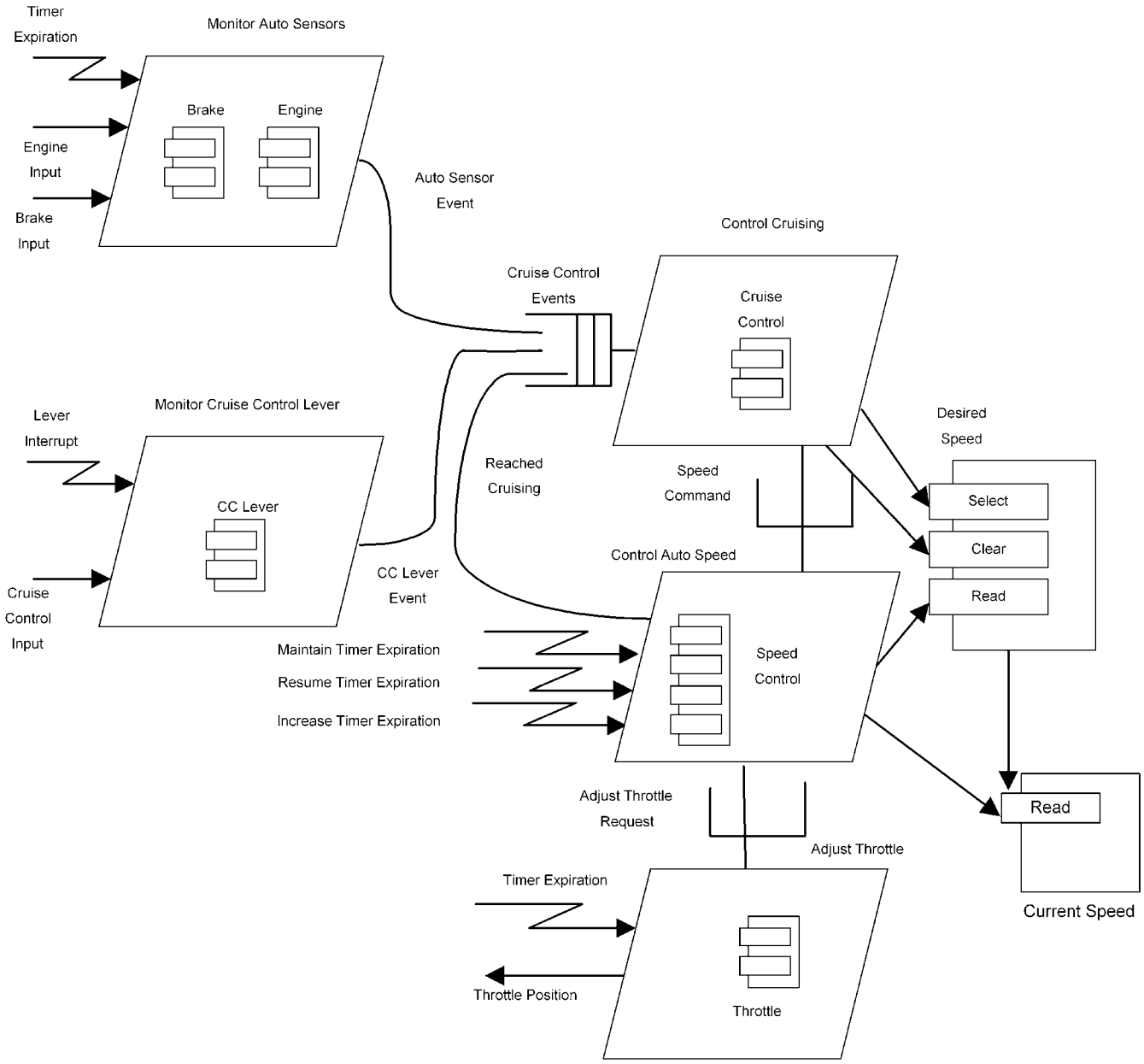


Fig. 16. A concurrent design generated by CODA from the data-flow diagram fragment given in Fig. 15.

instance. The query interface sends an explain message to the appropriate design element, which then writes its design history to the display. For each decision made about the design element, the design element reports the rule used to make the decision, the action taken, and the rationale for the action. In addition, a designer can request that a decision trace file be generated during a design session. In this case, a record of all design decisions is also available immediately after the design has been generated.

6 Case Study: Automobile Cruise-Control Subsystem

To illustrate the practical application of the ideas discussed in Section 4, this section describes how CODA generates a design for an automobile cruise-control subsystem. Only a small excerpt is given; the entire case study is described elsewhere [21]. In what follows, the CODA model analyzer transforms a fragment from a flow diagram model into a COBRA model by annotating the flow diagram with labels that represent semantic concepts from the COBRA meta-model discussed previously in Section 5.1. From the annotated flow diagram, the CODA design generator interacts with an experienced designer to produce a concurrent design. CODA is also capable of generating a design for a novice designer, who gives no guidance during the design generation. The 10 designs evaluated in Section 7 include designs generated both with and without human assistance.

6.1 Applying the CODA Model Analyzer

The data-flow model fragment in Fig. 15 uses RTSA notation, described in Fig. 1, to depict the fundamental speed control aspects of a cruise-control system. Fig. 15 also represents the

TABLE 2
Candidate Tasks Allocated by the CODA Design Generator

Candidate Tasks	Transformation	Structuring Criterion
Task 1	Increase Speed	Controlled Periodic Task
Task 2	Maintain Speed	Controlled Periodic Task
Task 3	Resume Speed	Controlled Periodic Task
Task 4	Cruise Control Select Desired Speed Clear Desired Speed	Control Task Control Cohesion Control Cohesion
Task 5	Brake	Periodic Input/Output Task
Task 6	Engine	Periodic Input/Output Task
Task 7	Throttle	Periodic Input/Output Task
Task 8	Cruise Control Lever	Asynchronous Input/Output Task

output of the CODA model analyzer. Each RTSA syntactic element on the flow diagram is annotated with a label, in square brackets, which corresponds to one of the 36 COBRA semantic concepts listed in Fig. 4. (Disables, Enables, and Triggers are not annotated because the labels would be redundant.) Each annotation depicts the COBRA semantic concept assigned by the model analyzer to the associated syntactic element. The CODA model analyzer inferred the correct COBRA semantic concepts for every element of the flow diagram, consulting with the designer on only one point: whether terminators represented devices. When the designer is unable to classify the terminators, the model analyzer assumes, correctly in this case, that all terminators represent devices. In cases where the assumption proves incorrect, then the resulting design would also prove incorrect. The automated capture of design rationale helps the designer to identify instances where assumptions were used.

Once the semantic classifications are completed, the model analyzer checks each element to determine if additional information must be supplied. In Fig. 15, six event flows represent timers. The model analyzer ensures that the designer provides a positive period for each timer. The model analyzer also discovers a system input from an asynchronous device, the cruise control lever. The model analyzer asks the designer to provide a value for the maximum rate at which these inputs are expected to arrive. After all the necessary information has been obtained, the model analyzer checks the semantic classifications and the axioms for each element in the flow diagram. In this case, all of the concepts are properly classified and all the axioms are satisfied. Had discrepancies been detected, the designer would be required to correct them prior to invoking the CODA design generator.

6.2 Applying the CODA Design Generator

After analyzing the flow diagram in Fig. 15, an experienced designer decides to generate a concurrent design, beginning with task structuring. The designer chooses a target environment and then invokes the CODA design generator to structure tasks for the design. The resulting design, as generated by CODA, is shown in Fig. 16.

6.2.1 Structuring Tasks

The design generator begins by allocating candidate tasks for each of three transformations in Fig. 15, Maintain Speed, Resume Cruising, and Increase Speed. The design generator uses the CODARTS criterion for identifying control tasks to allocate additional candidate tasks based on another transformation, Cruise Control. The remaining tasks identified by the design generator result from device-interface objects. The design generator allocates a task from each of three device-interface objects, Brake, Engine, and Throttle, based on the CODARTS criterion for identifying periodic input/output tasks. The design generator also allocates a task from another device-interface object, Cruise Control Lever, based on the CODARTS criterion for identifying asynchronous input/output tasks.

Next, the design generator examines the remaining, unallocated transformations, in an effort to allocate them to appropriate tasks based upon CODARTS criteria for sequential and control cohesion or upon guidance elicited from the designer. In this case, the design generator needed no guidance from the designer. Table 2 shows the initial task structuring decisions made by the design generator during this decision-making process.

During the next decision-making process, the design generator examines the set of candidate tasks in an effort to combine tasks, where feasible. The design generator combines three tasks (1-3) based on mutual exclusion because none of the constituent transformations, Increase Speed, Maintain Speed, and Resume Speed, can execute simultaneously. The design generator combines the Brake and Engine tasks, Task 5 and Task 6, respectively, because these periodic input tasks operate with identical, 100 ms, periods. The final task structuring, generated by CODA, is given in Table 3.

6.2.2 Defining Task Interfaces

After structuring tasks, the designer decides to continue building the design by defining interfaces between the tasks. First, the design generator determines the external interfaces for each task. Incoming data flows from devices, outgoing data flows to devices, and incoming event flows from devices are allocated to inputs, outputs, and interrupts, respectively, for the appropriate tasks. Each timer event flow stimulating a task is mapped to a timer interface for the stimulated task. As a last step, all data and event flows exchanged between tasks are identified and marked for subsequent consideration.

Next, the design generator considers how event flows between pairs of tasks might be allocated. The design generator allocates event flows from the Monitor Auto

TABLE 3
Summary of Task Structuring Decisions Made by CODA

Task	Transformations Allocated	Structuring Criterion
Control Auto Speed	Increase Speed Maintain Speed Resume Speed	Controlled Periodic Internal Tasks Mutual Exclusion
Control Cruising	Cruise Control Select Desired Speed Clear Desired Speed	Control Task Control Cohesion
Monitor Auto Sensors	Brake Engine	Periodic Device I/O Tasks Temporal & Functional Cohesion
Adjust Throttle	Throttle	Periodic Device I/O Task
Monitor Cruise Control Lever	Cruise Control Lever	Asynchronous I/O Task

Sensors and Monitor Cruise Control Lever tasks to queued messages. These events flow into a state-transition diagram and, thus, none should be missed and their arrival order should be preserved. In addition, the two input tasks that generate these events should not be delayed waiting for the Control Cruising task to accept the events. The design generator maps all control flows from the Control Cruising task to the Control Auto Speed task onto a single tightly coupled message. The design generator selects this mapping because the Enable and Disable signals are transmitted during a state-transition and, thus, the sending task requires synchronization with the task receiving these control flows.

The design generator is less certain how to map the event, Reached Cruising, which flows from the task Control Auto Speed to the task Control Cruising. In general, this decision depends upon whether the sender of the event needs to synchronize with the receiver of the event. The design generator cannot determine if this is the case. Since the designer is using CODA in its experienced-designer mode, CODA asks the designer whether synchronization is required for this event. In this case, the designer says synchronization is not required, so the design generator maps the event onto a queued message.

After deciding how to map all the events that flow between tasks, the design generator next considers how to map all the data flows between pairs of tasks. In this case, three data flows must be considered. Each of these data flows is an instance of Throttle Value. As Fig. 15 shows, these data flows arrive at the Throttle transformation from three transformations, Maintain Speed, Resume Cruising, and Increase Speed. These three transformations have been combined into a single task that is separate from the task that controls the Throttle. The design generator, uncertain about the synchronization requirements for these data flows, consults the designer for additional information. The designer indicates that the sender and receiver must rendezvous around these data flows; subsequently, the design generator maps all three data flows to a single tightly coupled message from the task Speed Control to the task Adjust Throttle.

Next, the design generator recognizes that one task, Cruise Control, receives queued messages from multiple sources. Since the designer is using CODA in experienced mode, the design generator offers an opportunity to assign varying priorities to these messages. In this case study, the designer declines the offer. The design generator then examines the facilities available in the intended target environment and defines appropriate mechanisms for holding queued messages. Since the target environment provides message queuing services and tasks exchange queued messages at a single priority, the design generator allocates a first-in, first-out message queue for each task that receives queued messages. A summary of the task interfaces generated by CODA is given in Table 4.

6.2.3 Structuring Information-Hiding Modules

Before the design can be completed, the designer must apply information-hiding criteria to identify modules. The CODA design generator makes most of the module structuring decisions without consulting the designer; however, since the designer is operating in experienced mode, the design generator consults the designer in a few cases where the designer's insights might improve upon the decisions.

The design generator begins module structuring by considering which transformations and data stores should form the basis for candidate information-hiding modules. The design generator discovers three transformations to combine into a single, function-driver module, two data stores from which to allocate data-abstraction modules, one transformation that forms the basis for a state-transition module, and four transformations that lead to device-interface modules.

Next, the design generator attempts to allocate any unallocated transformations and data stores to existing or new modules. Two unallocated transformations, Select Desired Speed and Clear Desired Speed, are mapped to functions incorporated into an existing data-abstraction module, Desired Speed.

At this stage, the modules in the design are established and the design generator next constructs the operations and associated parameters required by each module. This occurs without consulting the designer. Table 5 provides a summary of the module structuring decisions made by CODA for this case study.

TABLE 4
Summary of Task Interface Decisions Made by CODA

Interface Element	Data/Control Flows Allocated	Allocation Criteria
Auto Sensor Event	Engine On, Engine Off, Brake Pressed, Brake Released	Queued Message
Monitor Auto Sensors	Engine Strobe and Brake Strobe	Timer
Engine Input	Engine Data	Input
Brake Input	Brake Data	Input
CC Lever Event	Accel, Cruise, Off, Resume	Queued Message
Lever Interrupt	Lever Strobe	Interrupt
Cruise Control Input	Cruise Control Data	Input
Speed Command	Enable/Disable Maintain Speed Enable/Disable Increase Speed Enable/Disable Resume Cruising	Tightly- Coupled Message
Reached Cruising	Reached Cruising	Queued Message
Maintain Timer	Maintain Strobe	Timer
Resume Timer	Resume Strobe	Timer
Increase Timer	Increase Strobe	Timer
Adjust Throttle Request	Throttle Value (3)	Tightly-Coupled Message
Adjust Throttle Timer	Throttle Strobe	Timer
Throttle Position	Throttle Position	Output

6.2.4 Integrating Tasks and Modules

Once task and module structuring are completed, the designer asks the CODA design generator to integrate these two views. The design generator first determines the logical placement of the eight modules, relative to the five tasks. Device-interface modules for unshared devices are placed within the tasks that access the associated device; so, for example, the Brake module and the Engine module go inside the task named Monitor Auto Sensors. Modules accessed by a single task, such as Speed Control, which is accessed only by the task Control Auto Speed, are placed within the accessing task. Modules accessed by multiple tasks, such as Desired Speed, are placed outside any task.

Once the relationships between tasks and modules are determined completely, the design generator examines possible connections between modules residing outside any task. Where an operation in one such module invokes an operation in another, the design generator establishes a relationship stating that the invoking operation requires the invoked operation. For each module that provides operations required by another module, the design generator creates a relationship indicating that the providing module serves the requiring module. For example, in this case study, an operation, Select, provided by the module Desired Speed, requires another operation, Read, provided by the module Current Speed. Current Speed, then, serves Desired Speed. All of these decisions are made without consulting the designer.

At this point the design is complete. When the designer requests that the design be written, the design generator constructs a specification and design history for each task and module.

7 Evaluation

The approach described in the preceding sections was mapped onto various knowledge representation techniques provided by an expert-system shell, CLIPS Version 6.0 [29], to produce CODA. CODA was then applied to four real-time problems that often appear in the literature: an automobile cruise control and monitoring system, a robot controller, an elevator control system, and a remote temperature sensor. For each problem, CODA was used to

TABLE 5
Summary of the Module Structuring Decisions Made by CODA

Module	Transformation / Data Store	Structuring Criteria
Control Auto Speed	Increase Speed Maintain Speed Resume Speed	State-Dependent, Function-Driver Module
Desired Speed	Desired Speed Clear Desired Speed Select Desired Speed	Data-Abstraction Module DAM Update Operation
Cruise Control	Cruise Control	State-Transition Module
Throttle	Throttle	Device-Interface Module
CC Lever	Cruise Control Lever	Device-Interface Module
Brake	Brake	Device-Interface Module
Engine	Engine	Device-Interface Module

TABLE 6
Automation Achieved by CODA during Task Structuring

Design Decision	Total	Unassisted	Assisted
All Task Structuring Decisions	189	182 (96%)	7 (4%)
Identify Tasks (using structuring criteria)	82	82 (100%)	0
Input/Output Tasks	35	35 (100%)	0
Internal Tasks	45	45 (100%)	0
Resource Monitor Tasks	2	2 (100%)	0
Merge Tasks (using cohesion criteria)	107	100 (94%)	7 (6%)
Control Cohesion	16	16 (100%)	0
Mutual Exclusion	6	6 (100%)	0
Sequential Cohesion	66	60 (91%)	6 (9%)
Task Inversion	8	8 (100%)	0
Temporal/Functional Cohesion	11	10 (91%)	1 (9%)

analyze a flow diagram model and then to generate one or more concurrent designs.

The following discussion evaluates the effectiveness of the knowledge-based approach to design generation, as embodied in CODA. Two questions are addressed. First, what degree of automation was achieved by CODA on the four real-time problems to which it was applied? Second, how do the designs generated by CODA, with and without human assistance, compare with designs produced by a human designer for the same problems?

7.1 Degree of Automation Achieved with CODA

CODA automates two aspects of the design process: model analysis and design generation. The degree of automation achieved for each of these aspects is considered in turn below.

7.1.1 Automation Achieved with the Model Analyzer

During model analysis, CODA classified 358 elements on data/control flow diagrams. Of these, 308 elements, or 86 percent, required no help from the designer: 290 elements or 81 percent, were classified automatically, while 18, or 5 percent, were data stores, which are directly represented using RTSA notation. The remaining 50 elements, or 14 percent, were classified after interaction between CODA and a designer. For 29 elements, or 8 percent, CODA asked the designer whether a terminator represented a device, external subsystem, or user role. The remaining classification decisions where CODA required help were split between two categories: For eight elements, or 2 percent, CODA made a tentative classification that the designer had to confirm or override, and, for the remaining 13 elements, or 4 percent, CODA required additional information from the designer in order to make a classification. Further information about the performance of the model analyzer can be found elsewhere [22].

7.1.2 Automation Achieved with the Design Generator

The CODA design generator was used to generate 10 concurrent designs from data/control flow diagrams, augmented by the model analyzer with labels representing COBRA semantic concepts. Of the 1,571 CODARTS design decisions required to generate the ten designs, 1,527, or 97 percent, were taken without human intervention. Tables 6, 7, 8, 9, and 10 provide further detail.

Table 6 contains a summary of the 189 task structuring decisions made by CODA, where 82 decisions involved identifying tasks using the task structuring criteria and 107 decisions involved reducing the number of tasks using the task cohesion criteria (Sections 3 and 4.4.1). Seven decisions, all involving task mergers, used human assistance. Six of these decisions involved assignment of data transformations to an existing task in cases where the flow diagram indicates multiple possibilities. In general, choosing an appropriate task requires application-specific knowledge about the functions that each data transformation performs. CODA also asked the designer for help before combining two tasks with harmonic periods. An experienced designer might be able to discern discrepancies in task priority in cases where tasks might otherwise be combined based on temporal and functional cohesion.

Table 7 summarizes the 479 decisions taken by CODA when defining interfaces between tasks. CODA made 95 percent, or 454, of the needed decisions without any assistance. These included all 143 decisions required to identify data flows and control flows exchanged between tasks. Allocating control flows proved relatively easy for CODA. Two of 130 control flow allocations used designer assistance. Both cases involved ambiguity about the synchronization required when tasks exchanged a control flow. CODA made relatively liberal use of designer aid to allocate 170 data flows. In 20 of the 170 cases, CODA asked the designer about synchronization requirements surrounding a data flow. Allocating queue interfaces also proved relatively easy. Here, CODA consulted with a designer in three cases, in order to determine message priorities.

Table 8 reveals the performance of CODA while structuring information-hiding modules. Here, CODA made 163 of the 175 decisions without assistance. Identifying modules required no designer help because the model analyzer had classified transformations on the flow diagrams so that the CODA design generator could easily identify modules. To reduce the number of modules, CODA turned to a designer for help in 12 of 81 cases. In one form or another, each of the 12 cases involved application-specific knowledge about the function of data transformations. In some cases, functional cohesion provided sufficient reason to combine modules. CODA has no insight into functional issues without consulting a designer.

TABLE 7
Automation Achieved by CODA during Task-Interface Definition

Design Decision	Total	Unassisted	Assisted
All Task Interface Decisions	479	454 (95%)	25 (5%)
Identify Inter-Task Exchanges	143	143 (100%)	0
Allocate Data Flows	170	150 (88%)	20 (12%)
Input	53	53 (100%)	0
Output	37	37 (100%)	0
Queued Message	52	42 (81%)	10 (19%)
Tightly Coupled Message	28	18 (64%)	10 (36%)
Allocate Control Flows	130	28 (99%)	2 (1%)
Hardware Interrupt	28	28 (100%)	0
Timer	38	38 (100%)	0
Software Interrupt	20	20 (100%)	0
Tightly Coupled Message	15	13 (87%)	2 (13%)
Queued Message	29	29 (100%)	0
Allocate Queue Interface	36	33 (92%)	3 (8%)
Assign Priorities	3	0	3 (100%)
Allocate Messages	33	33 (100%)	0

TABLE 8
Automation Achieved by CODA while Structuring Information-Hiding Modules

Design Decision	Total	Unaided	Assisted
All Module Structuring Decisions	175	163 (93%)	12 (7%)
Identify Modules	94	94 (100%)	0
Device-Interface Module (DIM)	53	53 (100%)	0
State-Transition Module (STM)	8	8 (100%)	0
Data-Abstraction Module (DAM)	29	29 (100%)	0
Algorithm-Hiding or Function-Driver Module (AHM or FDM)	4	4 (100%)	0
Reduce Modules	81	69 (85%)	12 (15%)
Operation of DAM	48	48 (100%)	0
Sequential/Functional Cohesion	25	17 (68%)	8 (32%)
Merge DAMs	8	4 (50%)	5 (50%)

TABLE 9
Automation Achieved by CODA when Defining Module Interfaces

Design Decision	Total	Unassisted	Assisted
All Module-Interface Definition Decisions	525	525 (100%)	0
Determine Module Operations	413	413 (100%)	0
DIM Operation	173	173 (100%)	0
STM Operation	8	8 (100%)	0
DAM Operation	48	48 (100%)	0
AHM or FDM Operation	124	124 (100%)	0
Operation Internal to Module	59	59 (100%)	0
Determine Additional Operation Parameters	112	112 (100%)	0

TABLE 10
Automation Achieved by CODA when Integrating Tasks and Modules

Design Decision	Total	Unassisted	Assisted
All Task and Module Integration Decisions	203	203 (100%)	0
Place Modules Relative to Execution Threads	109	109 (100%)	0
Generate Task to Module Calls	54	54 (100%)	0
Generate Module to Module Calls	40	40 (100%)	0

TABLE 11
Similarity among Designs

Design	Task Structuring	Task Interface Definition	Module Structuring	Module Interface Definition	Task & Module Integration	All Design Decisions
Cruise Control - Assisted CODA vs. Human	1.00 (48/48)	1.00 (68/68)	.98 (45/46)	.97 (123/127)	.98 (63/64)	.98 (347/353)
Robot Controller - Assisted CODA vs. Human	1.00 (17/17)	1.00 (54/54)	.95 (18/19)	.93 (63/68)	1.00 (16/16)	.97 (168/174)
Elevator Control System - Assisted CODA vs. Human	.95 (20/21)	.89 (33/37)	1.00 (15/15)	.96 (48/50)	1.00 (15/15)	.95 (131/138)
Remote Temperature Sensor - Assisted CODA vs. Human	.95 (19/20)	.89 (31/35)	N/A	N/A	N/A	.91 (50/55)
Cruise Control - Unassisted CODA vs. Human	1.00 (48/48)	.97 (66/69)	.91 (42/46)	.96 (122/127)	.92 (60/65)	.95 (338/355)

While structuring information-hiding modules, CODA also defines module interfaces. As shown in Table 9, CODA required no help to make the 525 decisions that were required to define module interfaces. This result occurred because CODA encodes a predetermined strategy to define module interfaces. A human designer might choose among a wide range of strategies to create module interfaces.

Table 10 reports the automation achieved by CODA when integrating the task and module views of concurrent designs. All 203 decisions required were made by CODA without assistance. The decisions accomplished three objectives. First, 109 decisions determined how modules would be placed relative to execution threads for the various tasks in the designs. Second, 54 decisions generated calls from tasks to specific operations in modules that were accessed by multiple threads. Finally, 40 decisions generated calls from operations in one shared module to specific operations in another shared module.

7.2 Comparison of Generated Designs with Human Designs

The four case studies reported in this paper consist of real-time problems, specified with text, data/control flow diagrams, and, where applicable, state-transition diagrams, taken from the literature. For each of these problems, at least one design, generated by a human designer, exists in the literature [7], [11]. This allows the solutions generated by CODA to be compared with existing solutions from human designers. The designs generated by CODA exhibit minor differences from the human designs. Differences should certainly be expected because a human designer must make each and every decision when manually producing a design, whereas CODA attempts to minimize the interaction with the human designer by taking many decisions without consultation. In the case studies, differences appear where human designers take design decisions based on knowledge that CODA does not possess and cannot elicit, or where CODA takes predetermined strategies when a human designer might choose among a wide range of options. Table 11 presents a quantitative look at the similarity between designs generated by CODA and designs developed by human designers.

Each row of Table 11 reports the design decisions taken to produce two designs from the same specification. In each case, one design was produced by CODA, while the other design was produced by a human designer and reported in the literature. Rows one through four report results when CODA could consult an experienced designer for assistance. The fifth row documents the performance of CODA in a case where every design decision was taken without consulting a human designer. Columns two through six report the design decisions applicable to a particular phase of the design process. Column seven accumulates all the design decisions reported across columns two through six. Each cell contains a similarity metric, computed using the ratio shown in parentheses below the metric. A similarity value of 1.00 indicates two designs are identical; the lower the value, the more the designs differ. The following formula defines the similarity metric, S.

$$S = (\max(CD, HD) - \text{delta}(CD, HD)) / \max(CD, HD).$$

In the formula, CD denotes the number of design decisions executed by CODA, whether assisted or not, to generate the design. HD denotes the number of design decisions taken by the human designer to produce the design. The delta function denotes the number of design decisions that differ between CD and HD.

Considering all design decisions, the similarity between designs generated by CODA and those produced by a human ranged from a low of .91 for the remote temperature sensor to .98 for the cruise control. For the remote temperature sensor design, the human designers did not fully adhere to the CODARTS guidelines. Instead, the designers used an earlier version of the guidelines, called DARTS, which did not consider semantic interpretation of elements on the flow diagram model [10], [11], [30]. No module structuring information is reported for the remote temperature sensor because the human designers allocated

Ada™ packages rather than information-hiding modules; thus, these results cannot be compared legitimately with the module design generated by CODA.

In the absence of assistance, as reported in the last row of Table 11, CODA relies solely on its built-in knowledge to resolve ambiguous or incomplete situations. These situations generally fall into four classes:

1. decisions about merging tasks,
2. decisions about merging modules,
3. decisions about the synchronization requirements between tasks, and
4. decisions about assigning priorities to intertask messages.

The CODA knowledge base contains default rules that take conservative decisions, which lead to valid designs that can be less efficient than designs generated with human assistance. When working without human assistance, CODA tends to generate designs that contain more tasks and modules. In addition, without human input, CODA will generally map event and data flows to queued messages in order to avoid deadlocks and to enable tasks to execute freely whenever possible. In some runtime systems, queued messages can take longer to exchange than tightly coupled messages.

As the last row of Table 11 shows, even without assistance from an experienced designer, CODA can produce designs fairly similar (.95) to those produced solely by human designers. This positive result occurs because the cruise control specification was developed with strong adherence to the COBRA guidelines and because the specification contained just a few situations where CODA might profit from consulting a designer. In other cases, results obtained without human assistance can be expected to vary significantly depending on the number of decisions that CODA faces where a human designer might provide effective advice.

8 Discussion

In addition to the performance of CODA some other issues merit discussion. In Section 8.1, the approach embodied in CODA is compared against some other approaches that aim to automate software design methods. Section 8.2 gives a summary of contributions from the current work, as described in this paper.

8.1 Comparison of CODA with Other Approaches to Automate Software Design Methods

Table 12 compares CODA against the four approaches to design-method automation that were described earlier in Section 3.3. The table indicates some advances achieved by CODA. First, three methods, CAPO, EARA, and STES, produce sequential designs. While SARA produces a concurrent design, it does so by using a second behavioral model that restricts the parallelism present in the basic data flow diagram. CODA can produce a concurrent design from a single behavioral model, documented with a data/control flow diagram. Second, since CAPO, EARA, STES, and SARA use a single technique for design generation, either production rules or clustering algorithms, a number of desirable properties are difficult to realize in their design generators. For example, completeness and consistency checking, which cannot be implemented with clustering algorithms, can be implemented only with difficulty using rules. CODA uses knowledge-based queries to automatically check a generated design for completeness and consistency. Third, only two of the methods, EARA and CODA, provide explicit traceability from the behavioral model to the software design. For EARA, this traceability is strictly one-way, from specification to design. For CODA, the traceability is bidirectional. In addition, CODA alone automatically captures the rationale for design decisions. Fourth, although most of the approaches require elicitation of information from a human designer, only CODA provides distinct operating modes to accommodate experienced and inexperienced designers. Fifth, CODA alone can vary the generated designs to account for variations in the target environment. Significant variations might include the availability of message queuing services and the number of signals permitted between tasks. Many of the characteristics of a target environment become significant when constructing a concurrent design.

8.2 Contributions

Unlike previous approaches to design automation, the approach described and evaluated in this paper develops and exploits an underlying metamodel that can represent and reason about instances of requirements models, design models, and the relationships between the two. As a result, the approach leads to several improvements in the state-of-the-art in software design automation. First, the existence of a metamodel enables a designer to check design model instances for consistency and completeness with respect to the metamodel. Such consistency and completeness checking, not supported by previous design automation approaches, helps a designer to improve the quality of a design. Second, the metamodel enables the design automation system to explicitly and automatically track traceability between elements in a requirements model and elements in a design model. While a few design automation tools have previously supported traceability, the approach described in this paper provides bidirectional traceability at a finer level of granularity and also includes traceability constraints that can be used during automated consistency checking. More importantly, the approach discussed in this paper also automatically captures design rationale. Using the design rationale capture mechanism, a designer can review the reasoning behind various automated design decisions and can determine if any of the decisions need to be overridden.

Further, novice designers can use the rationale capture mechanism to learn the reasoning behind various steps and heuristics included in the design method that underlies the automated design mechanism. The approach can also compensate for novice designers. Specifically, when a designer is a self-declared novice, the design automation mechanism will not query the designer regarding various subtle issues likely to require the knowledge of an experienced designer. Instead, the design automation mechanism uses default assumptions to resolve these subtle issues without interacting with a novice. The

TABLE 12
CODA Compared with Other Automated Software Design Methods

Feature	CAPO	STES	EARA	SARA	CODA
Input Model	DFDs	DFDs	DFDs	DFDs/SVD	DFD/CFDs
Output Model	Structure Charts	Structure Charts	Structure Charts	GMB / Structural Model	Task/Module Specs/Design Meta-Model
Decision Method	Coupling/ Cohesion	Structured Design	Structured Design	Mapping Rules	COBRA/ CODARTS
Underlying Techniques	Clustering Algorithms	Production Rules	Rule Rewriting	Production Rules	Production Rules/ Semantic Modeling
Completeness/ Consistency Checking	No	No	No	No	Yes
Traceability	Implicit	Explicit	Implicit	Implicit	Explicit
Design Rationale Capture	No	No	No	No	Yes
Interacts with Designer	No	Yes	Yes	Yes	Yes
Has a Experienced Mode and Inexperienced Mode	No	No	No	No	Yes
Varies Design With Target Environment	No	No	No	No	To a limited extent

resulting designs will be correct, but can be suboptimal. When interacting with a self-declared experienced designer, the automation mechanism will pose questions regarding various subtle issues that may appear in particular designs. In these cases, the affected design decisions can be influenced by specific guidance elicited from the designer. In the absence of such guidance, the design automation mechanism remains prepared to use its default assumptions.

Finally, the design automation mechanism provides some capability to vary the generated designs after taking into account specific design guidelines or specific traits of the intended target environment. Examples of design guidelines might include a task-inversion threshold, priority-assignment algorithms, and task-allocation algorithms. Target environment traits of interest might include: 1) the existence or absence of shared memory, message passing mechanisms, and priority queues, 2) the maximum number of intertask signals and task priorities supported by a target operating system, and 3) the number of processors. As implemented, the automated design mechanism can account in a limited form for many of these factors. More research remains to exploit this information in later phases of the design automation. For example, generated designs might be instantiated automatically for specific target platforms and then the design could be simulated to assess its performance.

8.3 Performance

Beyond its novel aspects, the design automation approach also fares quite well when considering its performance in two ways: 1) degree of automation and 2) quality of designs. The automated design mechanism consists of two main phases: model analysis and design generation. The model analysis phase aims to classify all symbols on a requirements model (represented with data/control flow diagrams) and to assign semantic tags to those symbols. For the models analyzed, 86 percent of the elements were classified without help from the designer. Further information about the intent, construction, and performance of the model analyzer can be found elsewhere [22]. The automated design generator is the main topic of the current paper. Of 1,571 design decisions required to generate 10 concurrent designs, the design generator made 1,527, or 97 percent, without human assistance. Further, the quality of the generated designs was quite good, when compared against designs generated by human designers and documented in the literature.

9 Future Research

The work reported here suggests several directions for future research. One class of research directions addresses extensions to the current work; a second class investigates issues beyond the scope of the current work. Each class is addressed in turn.

9.1 Extensions to Current Work

The approach embodied in CODA might be extended to any software design methods that model behavior using graphical notations to represent data or control flow among semantic elements arrayed in directed graphs. For example,

the semantic concepts in COBRA appear quite similar in intent to stereotypes in the Unified Modeling Language (UML) [6], [31], [32], [33]. UML collaboration diagrams appear similar in conception to flow diagrams used in COBRA. UML collaboration diagrams, properly labeled with stereotypes and augmented with additional information, could possibly be input to a design generator to produce a concurrent design.

In another extension, future research might investigate the use of a design generator to automatically map a concurrent design onto specific hardware architectures. The work presented here addresses variations in target environments, but to a limited extent. For example, the message queuing and software signaling services provided by target environments are considered when CODA generates task interfaces. Larger issues, such as the number of processors and the availability of various forms of shared memory, have not been considered. In addition, various algorithms can be identified for assigning tasks to processors and for assigning priorities to tasks within each processor. This information might be exploited to support automated mapping to specific hardware.

In another extension, future research could investigate automated support for partitioning behavioral models into subsystems. The current research assumes that concurrent designs are generated for single subsystems only. Many concurrent designs are mapped onto distributed systems of networked computers. Designers use various criteria for allocating elements from behavioral models to subsystems that can be distributed onto loosely coupled computing nodes. These criteria might be automated.

Additional research might investigate the scalability of the approach to larger problems. Checking a behavioral model to ensure that all axioms are satisfied can take quite some time as the size of the model increases. In the current research, the largest problem tackled, an automobile cruise control and monitoring system, consisted of 58 nodes and 112 directed arcs. For this model, checking axioms took about 15 seconds on a 266 MHz Pentium II processor. Further research might establish the performance characteristics of the underlying model checking and design generation algorithms as the model size increases.

9.2 Beyond the Current Work

Evaluating the quality of the designs generated by CODA required comparison against the work of human designers on the same problems. This comparison leaves open the issue of the quality of designs in general. How can software designs, no matter what their source, be assessed for quality? The current work on CODA captures information about the frequency of task executions and about the maximum rate at which external stimuli arrive at the system. While not used in the current work, this information could facilitate future research regarding automated evaluation of the performance of designs using rate-monotonic scheduling theory or dynamic simulation. Evaluating design quality along other dimensions, such as maintainability, reliability, and testability, might also provide some interesting research challenges.

10 Conclusions

Advances in knowledge engineering hold potential for effective automation of software design methods. This paper presented a knowledge-based approach, integrating semantic data modeling with production rules and knowledge-based queries, to automate COBRA, an object-based behavioral modeling method, and CODARTS, a software design method for concurrent and real-time systems. The approach leads directly to an automated designer's assistant, CODA, which was applied to generate 10 designs for four real-time problems. During the generation of the designs for these case studies, the design generator made 97 percent of all design decisions without consultation. The remaining decisions involved a variety of cases calling for a designer's judgment. When defining module interfaces and integrating the task and module views of concurrent designs, CODA uses predetermined strategies to achieve full automation that leads to acceptable results. A human designer would be free to consider a wide range of options that might lead to designs exhibiting differences in detail from those generated by CODA. For the case studies, the similarity between designs generated by CODA and designs generated by human designers for the same problems varied from a low of .91 to a high of .98, where a similarity of 1.00 denotes identical designs.

The approach described in this paper could assist designers in creating concurrent designs. For example, CODA could be embedded in computer-aided software engineering (CASE) systems. Most CASE systems enable a designer to enter flow diagrams and structure charts, or other representations of a software design; however, a human designer, outside the CASE system and without automated assistance, must perform the process of creating the software design from the flow diagrams. Where a tool such as CODA is available, a designer could enter a flow diagram into a CASE system and then invoke automated assistance to generate a concurrent design. Such automation can capture design decisions and rationale and can maintain traceability between elements on the flow diagram and components in the design. The CASE tool might also store designer decisions and reuse them when the design is changed.

Acknowledgments

We gratefully acknowledge the financial support provided by the National Institute of Standards and Technology (NIST) and, most particularly, James H. Burrows. Without his support, the work reported in this paper could not have been carried out. In addition, we appreciate the insightful suggestions from Paul Black and Dolores Wallace, who provided substantive technical comments on the draft manuscript as part of the rigorous NIST process for reviewing technical and scientific manuscripts prior to submitting them for consideration by conferences and journals. We also appreciate the encouragement provided by David Tennenhouse, who reminded us of our responsibility to share the results of our work. Finally, if readers find the manuscript to be clear and interesting, then much of the credit belongs to the anonymous reviewers whose careful reading and significant suggestions showed us how to improve the paper. Of course, we take responsibility for any errors or flaws remaining in the paper.

REFERENCES

[1] P. Freeman, "The Nature of Design," *Tutorial on Software Design Techniques,* Freeman and Wasserman, ed., pp. 46-53, Apr. 1980.
[2] G. Booch, "Object-Oriented Development," *Trans. Software Eng.,* pp. 211-221, Feb. 1986.
[3] G. Booch, *Object Oriented Design With Applications.* Redwood City, Calif.: Benjamin/Cummings, 1991.
[4] P. Coad and E. Yourdon, *Object-Oriented Analysis.* Englewood Cliffs, N.J.: Yourdon Press, 1991.
[5] T. DeMarco, *Structured Analysis and System Specification.* Englewood Cliffs, N.J.: Prentice-Hall, 1978.
[6] M. Fowler, *UML Distilled.* with K. Scott, Addison-Wesley, 1997.
[7] H. Gomaa, *Software Design Methods for Concurrent and Real-Time Systems.* Reading Mass.: Addision-Wesley, 1993.
[8] D. Hatley and I. Pirbhai, *Strategies for Real-Time System Specification.* New York: Dorset House, 1988.
[9] M. Jackson, *System Development.* Englewood Cliffs, N.J.: Prentice-Hall, 1983.
[10] K. Nielsen and K. Shumate, "Designing Large Real-Time Systems With Ada," *Comm. ACM,* pp. 695-715, Aug. 1987.
[11] K. Nielsen and K. Shumate, *Designing Large Real-Time Systems With Ada,* New York: McGraw-Hill, 1988.
[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design.* Englewood Cliffs, N.J.: Prentice-Hall, 1991.
[13] S. Shlaer and S.J. Mellor, *Object Lifecycles—Modeling the World in States,* Englewood Cliffs, N.J.: Yourdon Press, 1992.
[14] P. Ward and S. Mellor, *Structured Development for Real-time Systems.* Englewood Cliffs, N.J.: Prentice-Hall, 1985.
[15] E. Yourdon and L.L. Constantine, *Structured Design,* Englewood Cliffs, N.J.: Prentice-Hall, 1979.
[16] J. Karimi and B.R. Konsynski, "An Automated Software Design Assistant," *Trans. Software Eng.,* pp. 194-210, Feb. 1988.
[17] J.P. Tsai and J.C. Ridge, "Intelligent Support for Specifications Transformation," *IEEE Software,* pp. 28-35, Nov. 1988.
[18] G. Boloix, P.G. Sorenson, and J.P. Tremblay, "Transformations Using a Metasystem Approach to Software Development," *Software Eng. J.,* pp. 425-437, Nov. 1992.
[19] K.E. Lor and D.M. Berry, "Automatic Synthesis of SARA Design Models From Systems Requirements," *Trans. Software Eng.,* pp. 1229-1240, Dec. 1991.
[20] D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Comm. ACM,* pp. 1053-1058, Dec. 1972.
[21] K. Mills, "Automated Generation of Concurrent Designs For Real-Time Software," PhD dissertation, George Mason Univ., Va., 1996.
[22] K. Mills and H. Gomaa, "A Knowledge-Based Method for Inferring Semantic Concepts from Visual Models of System Behavior," *ACM Trans. Software Eng. and Methodology,* vol. 9, no. 3, pp. 306-337, July 2000.
[23] R. Fikes and T. Kehler, "The Role of Frame-Based Representation In Reasoning," *Comm. ACM,* vol. 28, no. 9, pp. 904-920 Sept. 1985.
[24] E. Lim and V. Cherkassky, "Semantic Networks and Associative Databases," *IEEE Expert,* pp. 31-40, Aug. 1992.
[25] M.R. Genesereth and M.L. Ginsberg, "Logic Programming," *Comm. ACM,* vol. 28, no. 9, pp. 933-941, Sept. 1985.
[26] R.J. Brachman and J.G. Schmolze, "An Overview of the KL-ONE Knowledge Representation System," *Readings in Artificial Intelligence and Databases,* J. Mylopoulos and M.L. Brodie, eds., pp. 207-229, 1989.
[27] R.S. Michalski, "Pattern Recognition as Rule-Guided Inductive Inference," *Trans. Pattern Analysis and Machine Intelligence,* vol. 2, no. 4, 1980.
[28] F. Hayes-Roth, "Ruled-Based Systems," *Comm. ACM,* vol. 28, no. 9, pp. 921-932, Sept. 1985.
[29] National Aeronautics and Space Administration (NASA), Software Technology Branch, *CLIPS Reference Manual,* version 6.0. June 1993.
[30] H. Gomaa, "A Software Design Method for Real Time Systems," *Comm. ACM,* pp. 938-949, Sept. 1984.
[31] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide.* Reading, Mass.: Addison Wesley, 1998.
[32] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML.* Boston, Mass.: Addison Wesley, 2000.
[33] H.E. Eriksson and M. Penker, *UML Toolkit,* Wiley Computer, 1998.
[34] D. Barstow, "Domain-Specific Automatic Programming," *Trans. Software Eng.,* pp. 1321-1336, Nov. 1985.
[35] D. Barstow, "Automatic Programming for Device-Control Software," *Automating Software Design,* M.R. Lowry and R.D. McCartney eds., pp. 123-1140, 1991.
[36] E. Kant, F. Daube, W. MacGregor, and J. Wald, "Scientific Programming by Automated Synthesis," *Automating Software Design,* M.R. Lowry and R.D. McCartney eds., pp. 141-168, 1991.
[37] D. Setliff, "On the Automatic Selection of Data Structure and Algorithms," *Automating Software Design,* M.R. Lowry and R.D. McCartney eds., pp. 207-226, 1991.
[38] D.R. Smith, "KIDS—A Knowledge-Based Software Development System," *Automating Software Design,* M.R. Lowry and R.D. McCartney eds., pp. 483-514, 1991.
[39] D. Marques, G. Dallemagne, G. Klinker, J. McDermott, and D. Tung, "Easy Programming Empowering People to Build Their Own Applications," *IEEE Expert,* pp. 16-29, June 1992.
[40] S. Fickas and R. Helm, "Knowledge Representation and Reasoning in the Design of Composite Systems," *IEEE Trans. Software Eng.,* pp. 47-482, June 1992.
[41] S. Fickas and R. Helm, "A Transformational Approach to Composite System Specification," Technical Report CIS-TR-90-19, Univ. of Oregon, Nov. 1990.
[42] G. Fischer, A. Girgensohn, K. Nakakoji, and D. Redmiles, "Supporting Software Designers with Integrated Domain-Oriented Design Environments," *IEEE Trans. Software Eng.,* pp. 511-522, June 1992.
[43] C. Rich and R.C. Waters, "The Programmer's Apprentice: A Research Overview," *Computer,* pp. 10-25, Nov. 1988.
[44] C. Rich and R.C. Waters, "Automatic Programming: Myths and Prospects," *Computer,* pp. 40-51, Aug. 1988.
[45] C. Rich and Y. Feldman, "Seven Layers of Knowledge Representation and Reasoning in Support of Software Development," *IEEE Trans. Software Eng.,* pp. 451-469, June 1992.
[46] R. Waters and Y. Tan, "Toward a Design Apprentice: Supporting Reuse and Evolution in Software Design," *Software Eng. Notes,* pp. 33-44, Apr. 1991.
[47] G. Estrin, R.S. Fenchel, R.R. Razouk, and M.K. Vernon, "SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems," *IEEE Trans. Software Eng.,* pp. 293-311, Feb. 1986.
[48] H. Gomaa, "Software Development of Real Time Systems," *Comm. ACM,* vol. 29, no. 7, pp. 657-668, July 1986.
[49] H. Gomaa, "Structuring Criteria for Real Time System Design," *Proc. 11th Int'l Conf. Software Eng.,* 1989.
[50] M. Cochran and H. Gomaa, "Validating the ADARTS Software Design Method for Real-Time Systems," *Proc. TriAda Conf.,* Oct. 1991.

Kevin L. Mills holds advanced degrees in computer and information sciences, including a PhD in information technology from the George Mason University, Fairfax, Virginia. He is a research computer scientist in the Information Technology Laboratory at the National Institute of Standards and Technology. His current research investigates metrics and methods for quantitative measurement of software for distributed, ad hoc network services, and the measurement and modeling of distributed software systems and networks. He is a senior member of the IEEE.

Hassan Gomaa is a professor of software engineering in the Department of Information and Software Engineering at George Mason University, Fairfax, Virginia. He holds a PhD degree in computer science from Imperial College, London. He has over 25 years experience in software engineering, in both industry and academia, and has published more than 100 technical papers and two textbooks. He is a member of the IEEE and the IEEE Computer Society.