Knowledge-Based Software Architectures: Acquisition, Specification, and Verification

Jeffrey J.P. Tsai, *Fellow*, *IEEE*, Alan Liu, *Member*, *IEEE*,
Eric Juan, *Member*, *IEEE*, and Avinash Sahay

Abstract—The concept of knowledge-based software architecture has recently emerged as a new way to improve our ability to effectively construct and maintain complex large-scale software systems. Under this new paradigm, software engineers are able to do evolutionary design of complex systems through architecture specification, design rationale capture, architecture validation and verification, and architecture transformation. This paper surveys some of the important techniques that have been developed to support these activities. In particular, we are interested in knowledge/requirement acquisition and analysis. We survey some tools that use the knowledge-based approach to solve these problems. We also discuss various software architecture styles, architecture description languages (ADLs), and features of ADLs that help build better software systems. We then compare various ADLs based on these features. The efficient methods that were developed for verification, validation, and high assurance of architectures are also discussed. Based on our survey results, we give a basis for comparing the various knowledge-based systems and list these comparisons in the form of a table.

Index Terms—Knowledge-based system, software architecture, knowledge acquisition, architecture specification language, architecture style, formal verification, compositional verification.

1 Introduction

In the late 1960s, software engineers and system designers were faced with what was then termed the "software crisis." This crisis was the direct result of the introduction of a new generation of computer hardware. The new computers were substantially more powerful than hardware available up until then, making large applications and software systems feasible. However, the strategies and skills employed in designing software for the new systems did not match the new hardware capabilities. The results were delayed projects (often for years), considerable cost overruns, and unreliable and poorly performing applications. The need arose for new techniques and methodologies to design large-scale software systems.

The methodology proposed as an answer to the software crisis was the Waterfall paradigm. This methodology was first proposed by Royce [1] and later modified by Boehm [2], Jensen and Tonies [3], and others. The waterfall paradigm has seen many variations, amendments, and deletions, but all these share the basic assumption that software is developed in a sequence of distinct stages, namely requirements phase, design and coding phase, validation and verification phase, operation and maintenance validation phase.

The main assumption of the waterfall paradigm is that it begins with well-understood requirements and that those requirements and thus the design specification, are fixed. Tools supporting this paradigm usually enforce this rigidity by static type-checking and interface descriptions. In practice, though, standard techniques do not allow one to arrive at exact specifications. Often, the user does not know, and cannot anticipate his exact requirements. (The user's informal requirements description is more aspiration than specification.) Since the user has no experience on which to ground those aspirations, it is only by exploring the properties of some putative solutions that the user can find out what is really needed [4].

These problems are serious, and many critics have called for a replacement of the software life-cycle methodologies by new paradigms for software engineering. Three new methodologies emerged:

1) rapid prototyping,
2) executable specification, and
3) transformational implementation.

In rapid prototyping, a working model of the system is built for requirement validation. In the second approach, a system model is built which is executed to generate the system behavior in early stages of the software development. In transformational implementation, a concrete system is derived from a series of transformations, using automated support. By using these approaches, the system behavior can be observed and validated at early stages.

However, as the user gains a better understanding of his needs, the requirements need to be refined. The system developers have to handle the informal and incomplete user requirements and at the same time, maintain and evolve a consistent and formal internal representation of the user

• *J.J.P. Tsai, A. Liu, E. Juan, and A. Sahay are with the Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, IL 60607. E-mail: tsai@eecs.uic.edu.*

Manuscript received 15 May 1997; revised 27 Aug. 1998.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 108313.

1041-4347/99/$10.00 © 1999 IEEE

requirements. For this, every stage of the software life-cycle model must be formalized and a knowledge base for each stage should be developed. The representation of the knowledge base should be extensible and should support inferencing under various logic semantics (like, non-monotonic logic). The development systems must also support correct transformation and refinement of constructs across different levels of abstractions [5].

As computers and networking technologies are getting more powerful and cheaper, more and more application softwares have been developed. Many industries, such as telecommunications, avionics, banking, hospital, automotive, semiconductors, oil, pharmaceuticals, are all highly dependent on computers for their basic functioning. However, the techniques and tools to design and maintain complex software systems lag behind the growth of size and complexity of those systems [6].

Recently, knowledge-based software architecture paradigms have been proposed as better ways to support component-based technology. Software architectures bridge the gap between the user's needs and the desired solution (target system). At this level of design, the specification of the overall system structure (composed of components, communication protocols and control structures) is more significant than the low level algorithms and computations. This gives a correspondence between the system requirements and the elements of the constructed system. Some architectures support extensive typed systems in which the elements are represented in a hierarchy of types. This supports evolution through reuse, refinement and instantiation of the architectural elements.

Based on this new software paradigm, the development system may consist of the following sequence of phases:

- *knowledge/requirement acquisition,*
- *knowledge/requirements specification,*
- *architecture description in an ADL,*
- *validation and verification for high assurance,* and
- *the transformation to implementation level code upon which the target system is built* (Fig. 1).

The knowledge-base assistance is provided to the formal requirements, ADL and validation phases. As the system is validated for high assurance, the requirements and ADL description are incrementally refined. This feature supports the specification evolution. In this paper, important techniques developed to support the knowledge-based software architecture paradigm, namely, knowledge/requirement acquisition, architecture description languages, and verification will be surveyed. (Readers can refer to [93] for the related transformation techniques.)

2 Knowledge/Requirements Acquisition Systems

Software design is a knowledge intensive activity which requires a great amount of knowledge from different sources. In each phase of software life-cycle, there are specialists responsible for performing their tasks involving the clients who are usually the experts in the problem domains, but might have no idea in software development. The tremendous quantity of information is collected and processed by these specialists and used in these phases. In order to

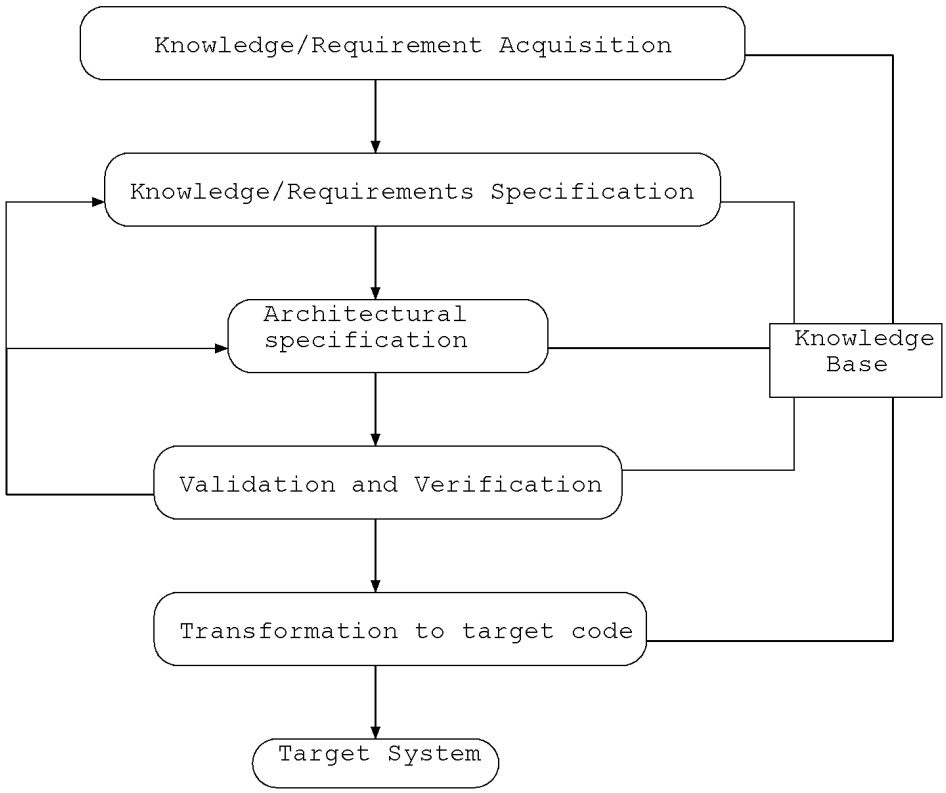

Fig. 1. The system organization.

assist these specialists, there are many knowledge-based software engineering tools introduced both in the market and in the research community. In this section, we survey and compare some of important knowledge/requirement acquisition systems. From the survey, we can get a better understanding on the techniques and models used by various knowledge-based systems.

2.1 ARIES

ARIES [7] is an environment for supporting analysts in modeling target domains and in entering and formalizing system requirements. It can be viewed as applying the notion of a presentation architecture to the domain of software engineering and incorporating a strong coupling to a transformation system. The key feature of the approach is to have a single highly expressive underlying representation with notations of differing degrees of expressiveness. The process of constructing a requirements specification is viewed as

- *acquisition*,
- *reasoning*,
- *evolution*, and
- *presentation*.

A modularized central repository approach is used for gathering together of all relevant requirements information into a common, shared framework [8]. The structure called *folder* has a set of declarations (type, relation, event, instance, and invariant declarations). The reusable folders are organized into hierarchies according to their degree of specificity to a particular task. A single, highly expressive knowledge representation scheme places a heavy emphasis on organization and use of domain knowledge in requirements analysis. Analysis is performed to uncover errors in the specification and fix them. The interface and the simulator for ARIES have also been developed [9], [10].

2.2 Attempto

Attempto is a system which translates specification written in Attempto Controlled English (ACE) into discourse representation structures and Prolog [11]. The purpose of ACE is to provide a formal specification without requiring the user to have a training in a formal language. ACE is a subset of English and has nouns, verbs, adjectives, numbers, and technical symbols as vocabulary. Declarative sentences are used as the basic construct of the specifications. Attempto is a knowledge-based system which provides lexical editor and a translator. The lexical editor interactively allows the user to modify and extend the lexicon incrementally. The knowledge base contains the translated form of the specification and allows the user to query the knowledge base. The knowledge base is also used for simulation and prototyping purposes.

2.3 Critter

It is an interactive model of composite system design incorporating deficiency-driven design, formal analysis, incremental design and rationalization, and design reuse [12]. Critter supplies knowledge of composite system design strategies and concepts; the designer supplies domain specific knowledge to validate these strategies and concepts in a particular application domain [13]. By using *off-the-shelf formal languages*, their goal is to solve the problems in the stage between the requirements acquisition process and the implementation process. They have used Petri nets for this purpose. The Critter model is the sum of the following components:

- *design state representation: possible behaviors and constraints*.
- *solution (consistent) state or leaf-node checker*.
- *move or design operators*: transforming one composite design state to another (analysis ⇒ deficiency ⇒ remedy).
- *heuristics for selecting design operators*: human designer.
- *tools*: a Petri-net editor, design components manager, browser, etc.

Associated to Critter, a system called OPIE, uses scenarios for analysis and modification of a specification [14].

2.4 JANUS

The JANUS system [15] implements the concept or domain-oriented design environments. The system uses an approach that embeds human-computer cooperative problem-solving tools into knowledge-based design environments that work in conjunction with human software designers in specific application domains [16]. The system is an integrated environment that can support the co-evolution of specification and construction. By integrating the upstream and downstream activities, designers are simultaneously articulating "what" they need to design and "how" to design it. It assists the user to locate design information, comprehend the retrieved information, and modify it according to their current needs. This idea is implemented in subsystems, called CATALOGEXPLORER, EXPLAINER, and MODIFIER. Through integration of cooperative problem-solving approaches with knowledge-based techniques, it is a conceptual framework for supporting co-evolution of problem specifications and software implementation that focuses on the role of human designers.

2.5 RAKES

RAKES is a requirements analysis system which interviews the clients in order to produce specifications for functional specification and nonfunctional specification [17]. It is included in the FRORL methodology [5], which uses the formal language, called FRORL [18], in every aspect of the software development phases. This helps in organizing the reusable components for the future specification in the same domain and gathering the domain knowledge from the specification. The same technique for organizing and retrieving the knowledge base can be used. This leads to simplifying maintenance.

The information collected in RAKES can be used throughout our software development methodology. The information is transformed to the knowledge base and it can help the developer understand the problems appearing in the latter phase by providing the user with the *background* of what has happened before.

2.6 KAOS

KAOS [19] provides a goal-directed approach to requirements acquisition. There are three components in this approach, namely:

1) the *conceptual model*, which provides metamodels that are abstractions for the requirements models to be acquired, including both functional and nonfunctional requirements;
2) *acquisition strategies*, which are the steps for acquiring components of the requirements model as instances of metamodel components; and
3) the *acquisition assistant*, which provides automated support in acquisition steps.

Higher level abstractions such as

- *goal*,
- *operationalization*,
- *ensuring action*,
- *agent*,
- *responsibility*, or
- *alternative assignment*

are the attributes considered as the conceptual models. The basic idea of the acquisition strategies is the *goal-directed traversal* of the metamodel graph. More detailed information is supplied from visiting the nodes in the graph. The steps gear toward identifying and refining

- *goals*,
- *agents*,
- *constraints*,
- *objects*, and
- *actions*.

The reuse of both metalevel and domain-level knowledge is also investigated.

2.7 KBMS

KBMS is a knowledge-based system for modeling software system specifications [20]. The end users can communicate with the computer system at the higher level of mathematical models with a simple, restricted vocabulary. Their goal is to acquire and transform a user's informal application knowledge into an accurate software application model, and the software model is based on a simple state transition framework. The information that they are interested in is

- *characteristics of components*,
- *ranges of values*,
- *relationships between the components*,
- *initial and final states of the system*, and
- *actions*.

It is carried out in two stages:

1) the requirements elicitation stage, and
2) the requirements modeling stage.

The system is supported by a knowledge base, which contains domain specific knowledge and software engineering knowledge. The inference mechanism carries out these four steps:

1) the *focus procedures* step,
2) the *filter procedures* step,
3) the *selection procedures* step, and
4) the *execution procedure* step.

The input to the system is entity descriptions and process definitions with their constraints, and the output is a structural and behavioral model of an application expressed in an English-like text.

2.8 LEAP

LEAP [21] is an asset-based synthesis system, and its assets are in the form of templates used to automatically synthesize source-level programming-language code. This approach allows software production to be incrementally automated and in large part produced by system engineers rather than software engineers. Using the templates and existing component descriptions, the developers interactively compose, elaborate, and refine descriptions on the basis of feedback from testing and analysis of partially developed designs. When a description is sufficiently detailed, LEAP can synthesize an implementation for that description as code in a high level design language, called the Common Intermediate Design Language (CIDL). LEAP then translates the CIDL code into Ada, C, or Lisp. Synthesis generally involves reusing existing code and constructing new code. As a description evolves, LEAP resynthesizes the implementation, automatically maintaining consistency between descriptions and their implementation.

The LEAP system descriptions offer relief from the well-known difficulty that users cannot write formal specifications. Composing a description from component descriptions allows well-understood component descriptions to be provided with complete implementable specifications. The user does not have to write these specifications, only modify them to accommodate specific application needs. LEAP has been used in small and medium size applications [22].

2.9 Marvel

The Marvel [23] software development environment uses a rule-based model of the development process. It provides automated help by applying forward and backward chaining among the rules, automatically invoking activities that are part of the development process. One distinguishing feature of Marvel is its integration of object-oriented data modeling and rule-based process modeling [24].

Marvel allows a project administrator to create and tailor an environment by defining a data model and a process model. Using the Marvel Strategy Language (MSL), the administrator writes specifications of the models and loads them into the kernel. Software artifacts (such as code and documentation) are abstracted as instances of classes, which are defined in the data model and stored in an object base. It allows so-called "structural" queries in the rules to traverse the object base structure. The query language consists of Boolean combinations of three structural primitives (Member, LinkTo, and Ancestor) and standard relational operators (<, =, and so on), which allow the query to navigate through the object base.

2.10 NATURE

NATURE is a project funded by the ESPRIT III program [25]. The program began in 1992, and the goal is to develop theories in knowledge representation, domain engineering, and process engineering. Prototypical models and tools are developed around Telos [26]. The experience from the DAIDA project [27] is expanded in this NATURE project.

The knowledge representation theory supports the combination of formal specification languages, semiformal methods, and informal aides for representing requirements. Formal languages like in the logic or algebraic forms offer preciseness and reasoning support. The semiformal methods, such as data flow or entity-relation diagrams, are good tools for communication between users and developers. Representations in text or pictures can be considered informal aides which are more familiar to the users.

The domain theory focuses on the cognitive studies of software engineering and semantic richness of specification languages. Two aspects of domain theory investigated are its basic structuring principles which supports abstraction and similarity-based reuse which organizes the descriptions of previous software components for later use. The problems to be considered are the granularity problem, coverage problem, the problems in acquiring, and modifying and classifying descriptions.

The process theory is based on the process models, such as activity-oriented models which find and execute a plan of actions leading to the solution, product-oriented models which represent the development process through the evolution of the product, and decision-oriented models which integrate the semantics attached to the evolutionary aspects of the design.

2.11 Process-Oriented Approach

A comprehensive framework for representing and using nonfunctional requirements during the development process has been proposed in [28]. The project emerged from the experience learned from the development of Telos, a language for representing knowledge about information systems [26]. Instead of evaluating the final product, the emphasis is on trying to rationalize the development process in terms of nonfunctional requirements.

Other than *accuracy requirements, performance requirements* are considered during the implementation phase when designs are mapped onto implementations [29]. Performance goals often focus on response time and throughput, and are developed for particular applications systems. A set of *intentional* operators are used in modeling the background information like "why." The nonfunctional requirements are well modeled, but the graph may result in complex goal-graph structures. Furthermore, it needs a theoretical foundation for representation and reasoning.

2.12 Requirements Apprentice

Requirements Apprentice (RA) [30] is the latest demonstration system for Programmer's Apprentice project. The purpose of the system is to gap between informal and formal specifications. By interacting with the user, it updates a requirements knowledge base and produces a requirements document. RA consists of three modules:

1) *Cake* [31], which is a knowledge representation and reasoning system containing truth maintenance, Boolean constraint propagation, equality, types, algebra, frames, and plan calculus;
2) *Executive*, which is an interface handling interaction with the user and providing high level control of the reasoning provided by Cake; and
3) *Cliche Library*, which is a collection of requirements relevant to the application domain.

The user starts with abstract requirements, and RA responses with its understanding. The user then incrementally define more requirements. Through these sessions, RA collects requirements from the user.

2.13 REMAP

The REMAP project [32] recognized the importance of capturing process knowledge to reason about the consequences of changing conditions and requirements in systems design and maintenance. The most important component of this process knowledge is the knowledge about reasons behind design decisions or design rationales which shape the design. The prototype has been implemented based on Telos and has been incorporated to a knowledge-based rapid prototyping system, called CAPS [33].

This model provides primitives to capture the knowledge about refinement, elaboration, and modification of initial requirements which lead to design solutions. The temporal knowledge keeps the information of versions of requirements specifications updated at different times and is useful in providing support in several areas including project management, design replay, and maintenance. Deductive rules can be used for modifying and inferring values of attributes.

In REMAP, a dependency network is used to report dependencies of constraints on design decisions as well as dependencies of design artifacts on constraints. With such an extended network, the belief status of a set of assumptions can be propagated to a set of design objects. They believe that changing requirements and assumptions necessitate changes to software systems, so changes to design rationale will automatically trigger changes in the belief status of design solutions, and this suggests redesign. As the dependencies among process knowledge and design artifacts are maintained by a reason maintenance system, a future revision in the belief status of relevant assumptions will automatically be propagated into changes in status of design solutions.

During various phases of the lifecycle, facilities to retrace the progression of steps that the design process went through can be beneficial. Design history information is useful in replaying the steps to facilitate understanding the evolution of the system as well as identifying the choice points where alternative decisions could lead to different solution paths. Design replay can be chronological or dependency directed.

2.14 RSML Tools

A set of tools are developed for analyzing the specifications written in RSML, which is a formal, state-based specification language [34]. The tools performs simulation and

analysis on the specifications which are based on function decomposition to check for completeness and consistency. The specifications are built on the features in RSML as states, decomposition, transition, and tables. The analysis is based on the compositional properties—union, serial, and parallel composition. The approach can be carried out in pieces and in an incremental fashion. Without the need of a large, global graph, each focused piece of specification can be analyzed thoroughly.

2.15 RTGIL Tools

The support tools for Real-Time Graphical Interval Logic (RTGIL) includes a graphical environment for a syntax-directed editor, an automated theorem prover, and a database and proof manager [35]. RTGIL is a propositional interval temporal logic for specifying real-time systems. The graphical editor assists the user in constructing a specification. The automated theorem prover uses a tableau form in checking the validity of the proofs while producing counter-examples. The database and proof manager is responsible for storing the valid proofs and reuse them efficiently. The types of properties supported are initial property, henceforth property, eventuality property, searches, and duration. The user is requested to select a set of premises to establish the theorem, and the tableau-theoretic method is carried out in the proof procedure.

2.16 SCR Tools

The automated support for the Software Cost Reduction (SCR) requirements method involves:

- a specification editor,
- a simulator, consistency checker, and
- a verifier [36].

Specifications are created and modified through the specification editor while the resulting specification is executed on the simulator. The specification then is checked by the consistency checker and the verifier for its properties. The SCR requirements method uses a formal language in a tabular form to specify the requirements. The properties that the consistency checker are interested include syntax, type, mode, value, reachability, disjointness, coverage, and circularity. The deterministic nature of the specifications allows the system to efficiently perform the static checks.

2.17 SPECIFIER

SPECIFIER [37] is a specification derivation system from the informal descriptions provided by the user, and it is an intelligent assistant to the requirements analyst interested in developing formal software specifications. It consists of

- a *preprocessor*, which accepts an informal definition of the problem expressed as a restricted subset of natural language,
- a *reasoner*, which produces a formal specification, and
- a *postprocessor*, which simplifies the axioms of the formal specification.

The approach can also be extended to software reuse [38].

A knowledge base mainly supports the reasoner for producing an informal specification by either *schemas* or *analogy*. A schema is an abstract representation of commonly occurring operations, along with the knowledge needed for instantiating it to particular cases. Analogy is a method which maps the current problem to the past results of similar problems maintained in the knowledge base. The formal specification languages used is based on a first-order theory and consists of

1) *precondition*,
2) *postcondition*, and
3) *required definitions*.

We summarize the above survey in this section by comparing the systems by the properties as follows:

Input the interface language between the user and the system
Output what the systems produce
Formal analysis degree of the formal analysis capability
Acquisition Method how the user requirements are collected
Where Used the phase in the software lifecycle, in which the system is used.

Table 1 shows the result of this comparison. The entry N.L. means *natural language*. From the table, we can see that there are three systems using general natural languages, five using structural natural languages (restricted natural languages), three using mixed representations, four using logic, and two using graphs for their input. As for output, there are seven systems producing a formal specification with accessories like natural languages or graphs, five producing just formal specifications (including Prolog), one producing a program, one for pseudo code, one for nets, and two producing just the analysis results. There are eleven systems which have some formal analysis capability. The methods are either incrementally building the requirements or interactively collecting the requirements.

3 Architecture Style and Architecture Description Languages

The component-based software architecture technique has gained a lot of importance these days. The architecture based development of software systems focuses on the architectural elements and their overall interconnection structure. The architecture of a system fills the gap between the high level box-and-line diagram and the low level programs used to implement the system. A system architecture is specified as having a set of components, connectors, a configuration and a set of constraints and is written in an Architecture Description Language (ADL). An ADL is the interface language between the user and the system. ADLs differ in terms of the aspects of the architecture that they can represent, tools they support for understanding and analyzing the architectural description and the overall level of support they provide to the system developers. Depending on the knowledge of the system that the user has, the user has to decide on the ADL to be used, so that the important aspects of the system can be better represented. Once the architectural information is captured from the user, the architecture is instantiated to the system that

TABLE 1
COMPARISON OF KNOWLEDGE/REQUIREMENTS ACQUISITION AND ANALYSIS SYSTEMS

Project Name	Input	Output	Formal Ana.	Acq. Meth.	Where Used
ARIES	restricted N.L.	N.L. Gist	Yes	Incremental	Requirements
Attempto	restricted N.L.	Prolog	No	Incremental	Requirements
Critter	Gist Nets	Gist Nets	weak	Incremental	Requirements Design
JANUS	N.L.	Code	No	Interactive	Requirements
KAOS	Frames Logic	Frames Logic	weak	Incremental	Requirements
KBMS	N.L.	Pseudo	No	Interactive	Requirements
LEAP	Graph	Graph Axiom	Yes	Incremental	Code Synthesis
NATURE	Mixed	Mixed	Yes	Interactive	Requirements
Marvel	Logic	Logic	No	Incremental	Code Synthesis
POA	Telos	Telos	No	Interactive	Non-Functional Requirements
RA	N.L.	Plan Calculus	weak	Incremental	Requirements
RAKES	restricted N.L.	FRORL Pseudo	Yes	Interactive Incremental	Requirements
REMAP	restricted N.L.	Nets & Telos	No	Incremental	Design
RSML	RSML	RSML	Yes	Incremental	Analysis
RTGIL	Graph	RTGIL	Yes	Interactive Incremental	Requirements
SCR	SCR	SCR	Yes	Incremental	Analysis
SPECIFIER	restricted N.L.	Logic Pseudo	weak	Interactive	Requirements

is to be implemented. The user can then verify that the implementation conforms to the specification and can analyze the various properties.

Many software systems exhibit commonalities in their architecture. Such systems are said to conform to a *style*. An architecture style represents a family of systems which share a common vocabulary of components, connectors and configurations, the underlying computation model, the semantic model and invariant properties. Some common architecture styles are pipe-filter, client-server, event-based, object-oriented, rule-based, hypertext, and layered systems [88]. Architecture styles guide in the design of large scale systems since most systems resemble one or more of these styles and the styles support the reuse of the style-specific solutions with well-defined properties and specialized analyses. The styles help in improved analysis and code reuse. It is also easy to understand the system organization once the standard styles are used. Another benefit is that the style-specific graphical and textual descriptions aid in the architectural design.

Pipe-filter style consists of components that read streams of data, apply local transformations to them and produce data on their outputs. The connectors (pipes) are simple conduits that transfer data from output of one filter to input of another. The pipes may be bounded or may restrict the type of data that can be transmitted through them. This style supports reuse, since two or more filters can be composed, and systems built in this style are easy to maintain and refine since filters can be replaced or new filters can be added with ease. Also, throughput and deadlock analysis can be performed on such systems.

In event-based architecture style, also known as implicit invocation systems, the components broadcast events in addition to communication via procedure calls. The connectors map event announcements by one component to procedure invocations in other components. An important property of this style is that a component does not know the identity of other components that react to its events. The style supports reuse and system evolution since new components can be added and old ones updated without modifying other elements in the architecture.

Object-oriented architectures help in building more flexible and extendible systems. Such architectures support description of systems at a level of abstraction closer to the problem domain and away from the low-level details. Since the individual objects of the architecture hide their implementation, they can be easily refined without affecting other architecture elements. But, if a component's (object) interface is changed, we need to modify all other components that invoke the services provided by this component.

Client-server architecture style is a kind of distributed processing system in which there are only two kinds of processes (components), a server and a set of clients. The server provides services to the clients, but may not know the identity of its clients. However, every client has the identity of the server to which it requests services.

Hierarchical layered systems is another instance of architecture styles based on call-return techniques. In this architecture style, each layer in the system provides services to the next higher layer and is provided services by the layer below it and is hidden from other layers. Operating systems and database systems use this architecture style. This style helps designing systems at many levels of abstraction. Also, it supports enhancement and reuse since modifying a layer affects only the immediate layers and different implementations can be used for a layer as long as it provides the same interface to its adjacent layers. However, for a system to be represented in this style, it has to be decomposed into multiple layers or levels of abstraction, which sometimes is a hard task.

Architecture description languages are used to support the architecture-based system development. An ADL is expected to exhibit the following features:

- ability to model components with property assertions,
- interfaces and implementations,
- ability to model connectors with protocols,
- property assertions and implementations,
- ability to model abstraction,
- encapsulation,
- types,
- type checking, and
- support for analysis.

Also for the ADL to be usable, it should support tools for understandable specification, multiple views, refinement, code generation, and dynamism [71].

Since the user requirements change from time to time, architecture components evolve. The ADL should support this evolution by allowing subtyping and refinement of components. The protocols for component interactions also keep changing. Hence, ADL should support reusability of connectors by allowing their subtyping and refinement. On treating these architecture elements as first-class objects, it becomes easy to reuse them as we can distinguish between types of these elements and their instances.

In the inheritance of element types, we can have both refinement and restriction of types. In case of restricted subtyping, certain properties of the component supertype are overridden, which can be handled if the ADL semantics can support nonmonotonicity of property assertions. Some of the ADLs that have been proposed for modeling domain-specific or general-purpose architectures are: Aesop [72], ArTek, C2 [79], Darwin [80], Unicon [71], Rapide [82], Wright [81], and MetaH [83]. We survey a few of them in this section.

In general, there are two important aspects that should be reflected in any ADL: It must have a simple, understandable syntax on one hand, and a formal syntax and semantics and analysis tools on the other hand. There have been attempts to identify important characteristics and requirements that an ADL should have [87]. Also, the basic features that need to be satisfied by all ADLs were proposed [87]. Shaw and Garlan [87] specifies the following kinds of features that an ADL should provide: abstraction, reusability, composition, configuration, heterogeneity, and analysis. This distinguishes ADLs from module interconnection languages, formal specification languages and programming languages. Clements [85] discusses the features that make ADLs different from requirements, programming and modeling languages.

While most of the ADLs provide an extensive support for modeling components as they treat them as first-class entities, the same is not true for connectors. Darwin, MetaH, and Rapide specify connectors as instances that cannot refined or reused in the future. Architectures may describe large complex systems that evolve over time. The changes to an architecture may not be predictable. The ADLs must support features for evolution and dynamism. Architectural configurations should be able to provide communication between people that participate in development of the system at different stages of development. Languages like Aesop, C2, Darwin, MetaH, Rapide, and Unicon support semantically sound graphical notations for architectural configurations.

The support for hierarchical composition is important when it is required to describe systems at different levels of granularity i.e., either explicit representation of component behaviors or behaviors hidden in individual components and connectors that are abstracted away. Since architectures are intended to represent systems at higher level of abstraction than the implementation modules, the relationship between elements of the architectural description and those of the resulting executable system may not be one-to-one [73]. This is because, changes made to the low-level elements cannot be easily mapped or traced back to the architectural elements. Also, scalability is another important issue. An ADL must be able to support development of systems that may grow in size.

Wright is an ADL that models interface points as ports, whose interaction semantics is specified in CSP. It has an extensible type system but doesn't support evolution of components. Wright models configurations as explicit attachments which aids evolution from partial specifications, but it provides no support for application families.

Unicon has a predefined, enumerated set of types, the component semantics is represented as event traces. The connector semantics is implicit in the connector type. The constraints are given as restrictions on type of players (component interface points) that can be used in a given

role (connector interface points). Unicon supports composition of configurations, system generation, scalability, but no evolution of configurations as it doesn't support partial architectures or application families.

Rapide is an event-based ADL used for defining and simulating behaviors of system architectures. The components in Rapide are characterized as interfaces and the interface points consist of "provides," "requires," "action," and "services." It has no explicit connectors, it only characterizes them via "connection components". The semantics are given in terms of partially ordered event sets. Rapide has an extensible type system, supports parameterization of types and inheritance through structural subtyping. It also allows the behavior associated with a specific type to be parameterized by specifying event patterns. It supports mapping between architectures at different levels of abstraction. Architectures can be mapped to interfaces, hence the support for composition. Again, in Rapide too, there is no support for partial architectures or application families. The events poset execution model allows modeling of timing information.

ACME [84] is an architecture interchange language used to map architectural specifications across different ADLs. It supports an extensible type system, with parameterization, for components and connectors (based on protocols). But it has no support for semantics for these elements, it can use other ADLs' semantic models. Attachments can be understood via explicit and concise textual specifications. ACME allows composition and heterogeneity, but no refinement. Evolution is possible due to explicit configurations, but there is no support for application families.

Darwin uses *π-calculus* to model the component interaction and composition properties. The connectors are specified as bindings but are not first-class entities (hence, no support for reuse, subtyping). Due to the hard-wired configurations, Darwin doesn't support evolution and scalability but allows composition of configurations.

C2 has extensible typed system for components and connectors. The interfacing is through messages and the causal relationships between input and output messages gives the semantics. It supports evolution of components by allowing name, interface and behavior subtyping. Dynamism is possible in terms of addition, deletion and reconfigurations.

Aesop [72] is a system for developing style-oriented architectural design environments. It is important to support architectural descriptions and analysis with tools and environments. Aesop system characterizes architecture styles as specializations of a generic object model through subtyping. This helps in generating style-specific design environments and hence, aid in construction of families of systems which come under the scope of that style. Wright provides explicit features for representation of different styles. Other ADLs like ArTek, Resolve, Unicon, and Demeter support certain generic features which can be used to indirectly achieve this capability.

OOADL [77] uses object-oriented paradigm as its backbone. The components in OOADL are characterized as objects which further decomposed into "internal behavior" and "architectural behavior" parts. The "internal behavior" part corresponds to the computation or action of the component with the semantics based on Z. It has no explicit connectors, connectors are implicitly characterized in the "architectural behavior" part with "export" and "import" specifications attached to it. This language provides "a_kind_of", "a_part_of", and "an_instance_of" keywords to represent the generalization, aggregation and instantiation relationships of the OO paradigm. Another feature of this language is that it provides a set of predefined most popular architecture structures which can be directly inherited by users and thus provides the refinement and reuse aspects of OO paradigm. But since, OOADL as no explicit connectors, no evolution for connectors is available and also nonfunctional properties are not supported for both components and connectors.

A system may be composed of multiple styles, in which case it becomes a heterogeneous architecture. The heterogeneous composition of architectures causes several problems since each style has its own properties, there is no common representation language for specifying the various constraints and attributes, and there is no unique way to compose these different styles. This issue has been addressed and some solutions proposed in [86]. Clements [85] compares various ADLs on the basis of:

- ability to represent styles;
- ability to handle dynamic architectures;
- ability to handle real-time issues; and
- support for creation, validation, refinement, analysis and support for building applications.

In this paper, we summarize our survey using the criterion, such as *understandability*, *composition*, *scalability*, *refinement*, and *evolution*, as shown in Table 2. In the table, by the term "hard-wired" we mean that the connectors are specified as instances and hence, cannot be refined or reused later.

4 Verification of Knowledge-Based Software Architectures

Studies have repeatedly shown that most of the cost of software development stems from design (or requirements) defects. Defects in design can cost hundred times more to fix in testing and maintenance phases than in the design phase. If we can identify those defects in the early stage of the software life-cycle, then we can eliminate these problems earlier and significantly reduce the cost of debugging, maintenance, and redevelopment. In addition the correctness of high assurance systems must be verified in order to avoid disastrous consequences.

Testing and debugging involves the process of detecting, locating, analyzing, isolating, and correcting suspected faults. Testing uses the runtime information of a program to examine its execution behavior. However, testing is not sufficient to prove the correctness of systems. In contrast, static analysis provides a methodology to verify the correctness of systems. In general, static analysis is supported by formalisms to specify the system precisely. Formal verification

TABLE 2
COMPARISON OF ADLS SURVEYED

ADL	Understandability of Specifications	Composition	Scalability	Refinement	Evolution
Wright	explicit textual	No	Yes,but fixed number of roles	No	partial archs. can evolve
Unicon	explicit textual and graphical	Yes	Yes	Yes	possible due to explicit configs.
Rapide	provides graphical notation	can map archs. to interfaces	No	Yes,can map archs. at different levels	No, due to hard-wired configs.
Darwin	provides graphical notation	Yes	No	Yes	No, due to hard-wired configs.
C2	explicit textual and graphical	Yes	Yes	No	partial archs. can evolve
ACME	explicit textual	Yes, using templates	Yes,but fixed number of roles	No	possible due to explicit configs.
Aesop	explicit graphical	Yes	Yes,but fixed number of roles	No	possible due to explicit configs.

methods are then applied to prove the logical correctness of the developed system with respect to the specification. However, most formal verification methods suffer from state explosion problem. Current research focus is how to deal with this problem. We will present and compare various methods in this section.

4.1 Formal Verification Methods

State space (or reachability) analysis provides a promising and automated method for the static analysis and verification of systems. However because of the complexity of the state-space explosion, efficient analysis by state space is restricted to small system models. With increasing computing power, larger and more complex systems are continuously developed. Techniques and tools need to be developed to analyze on large-scale systems. However, formal verification method is more applicable to architecture level of a software system since this level is closer to the problem domain and more abstract. Many techniques have been proposed to cope with the state explosion problem. *Equivalence* is the main concept shared in these techniques. Two system models are considered to be *equivalent* if no distinction can be made between them. Based on the concept of *equivalence*, it is possible to analyze a system model using another *equivalent* system model or state space. For example, Petri-net models can be analyzed using reachability graphs because Petri nets and reachability graphs are *equivalent* with respect to almost all dynamic behaviors and interesting properties. The notion of *equivalence* is very useful when we focus on particular behaviors / properties of systems. The following summarizes existing approaches to the state explosion problem.

Transformation of System Models. Many transformation techniques have been developed from the concept of *equivalence* in order to synthesize and / or reduce system models [51], [52]. A synthesized system model is *equivalent* to the original one. Therefore, the verification of synthesized system models can be eliminated. On the contrary, structural reduction is used to generate a reduced system model which is *equivalent* to the original one. Consequently, the analysis can be performed on a reduced system model. Synthesis and structural reduction of system models are powerful, but in some cases they are applicable only to special situations or special subclasses of specification formalisms.

Bit-Hash Technique. Holzmann [46] proposed a bit-hash technique to significantly reduce the memory requirement of state space analysis. In his technique, a hash function is employed to give a compact representation of state information. However, the analysis result may not be reliable because of hash collisions in some cases.

Symmetry Methods. Symmetry methods [44] exploit regularity of system structure and store mutually symmetric states as one. Symmetry methods require more complex analysis algorithms and cannot facilitate the analysis of nonsymmetric systems.

Symbolic Model Checking. Symbolic model checking [40], [41], [43] uses a fixed-point algorithm over a symbolic representation of state and transition relation (BDDs [39]). Symbolic model checking has been shown useful especially for the verification of hardware. However, its application to software systems may be affected by the effort in finding a compact encoding.

Partial Order Techniques. Partial order techniques investigate only a partial ordering of concurrent and independent events. The explored state space is *equivalent* in terms of certain properties, e.g., deadlocks and event occurrences. *Persistent* (or *stubborn*) *sets* [47] and *sleep sets* [45] are two main techniques proposed in the literature. Nevertheless, in the worse case, the state space remains exponential in the size of processes. A lot of research is focusing on the design of heuristic rules to reduce analysis complexity.

Compositional Verification Techniques. In compositional verification techniques, a system is considered as a collection of subsystems (processes). The main goal of compositional verification techniques is to hierarchically and efficiently generate *equivalent* and much smaller state spaces which are more amenable to analysis. There have been a considerable amount of studies on the compositional verification, especially in the area of process algebras [48], [49], [50]. However, current techniques are effective only for loosely coupled modules (i.e., subsystems with simple interfaces).

Incremental Verification Techniques. Many systems frequently undergo modifications to suite new user requirements or to satisfy the desired properties. Their specifications and hence, internal representations keep changing from time to time. However, due to the huge size of the state-space, it is very expensive to verify the requirements and constraints of the system after every small modification to the specification. The incremental approach is to make the cost of verification of the modified system proportional to the size of the change made to it rather than the size of the representation (the state-space) of the whole system [75], [76].

Hybrid Techniques. It has been suggested that more cost-effective techniques can be derived by integrating different verification techniques. For example, partial order techniques and structural reduction can be combined to further reduce analysis cost [67], symbolic state space search is combined with partial-order reduction methods for invariant checking [42], etc.

We summarize the above survey by comparing the formal verification methods as shown in Table 3. The comparison is made in terms of

1) the approach used to reduce state-spaces,
2) the applicability in general cases, and
3) their main drawbacks.

4.2 Compositional Verification

Among the formal verification techniques discussed above, compositional verification is considered more suitable for verifying component-based software systems or architecture specifications of a target system [78]. In this paper, we will focus on the important concepts and techniques in this approach. In compositional verification, a system is first divided into hierarchical modules (subsystems). State spaces of modules are then recursively constructed from reduced state spaces of submodules. As a reduced state space could be much smaller than the original one, the compositional verification technique provides an attractive approach for analyzing large-scale systems. Compositional verification techniques resemble structural-reduction techniques for system models [51]. Nevertheless, compositional verification techniques are applied to state spaces instead of system models. Compositional verification techniques are more effective than structural-reduction techniques since state spaces explicitly describe the dynamic behaviors of systems.

The main goal of compositional verification techniques is to hierarchically and efficiently generate *equivalent* and much smaller state spaces. However, the concept of *equivalence* does not consider compositional mechanisms. It is possible that the interesting properties of systems are not preserved after we replace one module with another *equivalent* module. In order to hierarchically generate an *equivalent* state space, the concept of *congruence* is essential. Two models m_i and m_i' are said to be *congruent* if for all systems $S = f(m_1, \ldots m_i \ldots, m_n)$ and $S' = f(m_1, \ldots m_i' \ldots, m_n)$, S and S' are *equivalent*, where f is an operator (or function). In other words, after one model is replaced with another *congruent* model, the new global state space is *equivalent* to the old one.

Various compositional verification techniques have been proposed for the algebraic treatment of processes by using equations and inequalities for processes expressions. These approaches to the algebraic treatment of processes can be identified by an acceptance of synchronized communication as the primitive means of interaction among processes [53]. Primary process algebras include for example, Milner's CCS [53], Hoare's CSP [49], and Bergstra and Klop's ACP [50]. *Equivalences* and *congruences* of processes are well founded in those process algebras for event-based systems.

Several works [68], [69], [64], [70] have addressed the general interest for asynchronous communication as the

TABLE 3
COMPARISON OF FORMAL VERIFICATION METHODS SURVEYED

Method Name	Approach	Applicability	Drawback
Transformation	Model reduction or synthesis	Weak	Limited application
Bit-hash	Packed state-spaces	Strong	Hash collisions
Symmetry	Packed state-spaces	Weak	Limited application
Symbolic	Packed state-spaces	Hardware	Difficulty in encoding
Partial order	Partial ordering	Strong	Larger state-spaces
Compositional	Abstraction	Strong	Simple interfaces
Incremental	Reuse data	Strong	No reduction

natural interaction of systems. In those formalisms, senders and receivers do not mutually synchronize their communication. The fact that output actions are nonblocking distinguishes those formalisms from primary process algebras. I/O automata and their variants [68] provide formalisms for modeling and verifying events systems which continuously receive input from and react to their environment. The distinguished character of IO-automata is the fairness assumption, i.e., there is no input-blocking in the model. In contrast, the assumption of input-blocking is used in some formalisms [69], [64], [70]. Boer et al. [69] proposed an abstract language (*L*) which is a generalized result obtained from several works. The language *L* had been applied to concurrent constraint programming [69]. In [64], [70], the semantic of nonblocking output actions is acquired using Petri nets whose compositionality is based on a fusion of places. Valmari [64] uses *CFFD equivalence* to derive *congruent* state spaces of modules (subsystems). Basten and Voorhoeve [70] presented an algebra semantics for hierarchical P/T nets in terms of an ACP-like algebra [50]. They also provided an equational theory for *weak bisimulation* (or *observable equivalence*) of hierarchical P/T nets.

We will discuss research issues and status of compositional verification techniques (except for I/O automata and their variants [68]) in the following sections.

4.2.1 Effectiveness of Compositional Verification Techniques

The analysis complexity of a formal verification tool decides it's usage in real applications. In the case of compositional verification, the execution time depends on the effectiveness and time complexity of condensation (reduction) algorithms. The time complexity of condensation algorithms determines how fast the state space of one module can be condensed. The effectiveness of condensation algorithms decides how well we can condense the state space of one module. The effectiveness of condensation algorithms also dramatically affects the execution time because the size of a composite state space is determined by the size (complexity) of condensed state spaces used in the composition.

Bisimilarity-based [53] and *failure*-based [49] equivalences are popular in contemporary process algebra verification of concurrent systems. The strength of *bisimilarity*-based equivalences is in their lower time complexity of reduction algorithms [63], while *failure*-based equivalences could lead to much smaller condensed state spaces. Since it is very difficult to preserve the state-oriented information using *bisimilarity*-based equivalences, we focus on *failure*-based equivalences technique in this paper.

In formal verification techniques—*Stable failure equivalence* [65], *CFFD equivalence* [64], *CSP-failure equivalence* [49], and *IOT-failure equivalence* [61]—are all defined in terms of trace semantics. Therefore, they all suffer from the PSPACE-complete problem [63], when we want to generate minimum state spaces. To avoid the high time-complexity of state-space condensation, rule-based methodology has been proposed in several research projects. For example, Sabnani et al. proposed three rules to reduce the number of states in a finite state machine, while maintaining its observational equivalence [56]; Shatz, Murata, et al. [57] had introduced several special reduction rules for Ada nets; Valmari introduced three rules for the compositional verification of deadlock [58]; Juan, Tsai, and Murata developed a set of condensation rules [59], [60], [61] based on *IOT-state equivalence*, *IOT-state equivalence*, and *firing-dependence theory* to support compositional verification. Most of their condensation rules resemble traditional reduction rules for Petri nets [51] in that rule preconditions and rule application consider only a small portion of the state space for condensation. The state space can be reduced after one or more rules succeed. The condensation rules in [59], [60], [61] are more effectiveness and lower overhead based on their experiments. They have also developed an automated analyzer, called IOTA, using their condensation rules which show the effectiveness and efficiency in analyzing large-scale systems.

4.2.2 State-Oriented Information

Current compositional verification techniques are developed mainly for even-based systems without preserving state-oriented information. The reachability property of states has been considered as a convenient way for analyzing critical conditions of systems, e.g., buffer overflow. In addition, the reachability property of states provides useful information for the understanding and modification of systems. The reachability property of states is even much more important in the practicality of compositional verification techniques. In compositional verification techniques, most of event occurrences are discarded in order to provide a small state space for analysis. Since most of analysis data (the event occurrences) are lost, it will be extremely difficult for developers to effectively debug and modify an improper system design. This difficulty could be minimized by using the reachability property of states, since a state indicates detailed conditions of the whole system.

In addition, state-oriented information provides a convenient way to distinguish different semantics of deadlock property. With the use of the deadlock property, we can identify deadlocking systems which cannot do anything. Nevertheless, we cannot tell whether a system fails or successfully terminates. Therefore, explicit representations of deadlock, divergence, and successful termination, are proposed for event-based systems [62]. In the IOTA system, they distinguish failure from successful termination by the conditions of deadlock, i.e., deadlock states. IOTA can generate a smaller state space because it is not necessary to retain the occurrences of successfully terminating events for vast processes in a large-scale system.

5 Conclusion and Future Research

In this paper, we have surveyed various knowledge- and requirements-acquisition systems. We also discussed some architecture styles and made a comparative study of various ADLs. Finally, we discussed some efficient verification techniques for high assurance of large system architectures. The important topics for further research in this area include:

- techniques and tools for the acquisition of design rationale and knowledge in a heterogenous, multimedia, distributed and mobile multiagents environment;
- the correctness and consistency of the captured requirements and design knowledge [89], [90];
- the interaction, communication, negotiation, and cooperation among various models and metamodels;
- the design of effective and efficient transformation systems [91], [92]; and
- scalable formal verification techniques.

ACKNOWLEDGMENTS

We thank the anonymous reviewer, whose comments have helped us to improve the presentation of the paper. Jeffrey J.P. Tsai was supported, in part, by the United States National Science Foundation and the U.S. Defense Advanced Research Projects Agency under Grant No. CCR-9633536; and by the U.S. Air Force Rome Laboratory under Grant No. F30602-95-1-0035.

REFERENCES

[1] W.W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," *Proc. Wescon*, 1970.
[2] B.W. Boehm, "Software Engineering," *IEEE Trans. Computer*, pp. 1,226-1,241, vol. 25, no. 12, 1976.
[3] R. Jensen and C. Tonies, *Software Eng.*, Prentice Hall, Englewood Cliffs, N.J., 1979.
[4] C.V. Ramamoorthy et al., "Software Engineering: Problems and Perspectives," *Computer*, pp. 191-209, 1984.
[5] J.J.P. Tsai and T. Weigert, *Knowledge-Based Software Development for Real-Time Distributed Systems*, World Scientific Publishing, Singapore, 1993.
[6] J.J.P. Tsai, Y. Bi, S.Y.H. Yang, and R.S. Smith, *Distributed Real-Time Systems*, Wiley and Sons, New York, 1996.
[7] W.L. Johnson, M.S. Feather, and D.R. Harris, "Representation and Presentation of Requirements Knowledge," *IEEE Trans. Software Eng.*, pp. 853–869, vol. 18, Oct. 1992.
[8] W.L. Johnson and M. Feather, "Building An Evolution Transformation Library," *Proc. 12th Int'l Conf. Software Eng.*, pp. 238–248, Nice, France, IEEE CS Press, Mar. 1990.
[9] W.L. Johnson, K.M. Benner, and D.R. Harris, "Developing Formal Specifications from Informal Requirements," *IEEE Expert*, pp. 82–90, vol. 8, Aug. 1993.
[10] K.M. Benner, "The ARIES Simulation Component (ASC)," *Proc. Eighth Knowledge-Based Software Eng. Conf.*, pp. 40–49, Chicago, Sept. 1993.
[11] N.E. Fuchs and R. Schwitter, "Attempto Controlled English ACE," *Proc. First Int'l Workshop on Controlled Language Applications*, Katholieke Universiteit Leuven, Belgium, Mar. 1996.
[12] S. Fickas and R. Helm, "Knowledge Representation and Reasoning in the Design of Composite Systems," *IEEE Trans. Software Eng.*, pp. 470–482, vol. 18, June 1992.
[13] S. Fickas and P. Nagarajan, "Critiquing Software Specifications," *IEEE Software*, pp. 37–47, vol. 5, Nov. 1988.
[14] J.S. Anderson and B. Durney, "Using Scenarios in Deficiency-Driven Requirements Engineering," *Proc. IEEE Int'l Symp. Requirements Eng.*, pp. 134–141, San Diego, Jan. 1992.
[15] G. Fischer, "Domain-Oriented Design Environments," *Proc. Seventh Knowledge-Based Software Eng. Conf.*, pp. 204–213, McLean, Va., Sept. 1992.
[16] 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, vol. 18, June 1992.
[17] A. Liu and J.J.P. Tsai, "A Method for Requirements Analysis and Knowledge Elicitation," *Int'l J. Artificial Intelligence Tools*, pp. 167–183, vol. 5, nos. 1/2, 1996.
[18] J.J.P. Tsai, T. Weigert, and H. Jang, "A Hybrid Knowledge Representation as a Basis of Requirements Specification and Specification Analysis," *IEEE Trans. Software Eng.*, pp. 1,076-1,100, vol. 18, no. 12, Dec. 1992.
[19] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-Directed Requirements Acquisition," *Science of Computer Programming*, pp. 3–50, vol. 20, Apr. 1993.
[20] K. Zeroual and P.-N. Robillard, "KBMS: A Knowledge-Based System for Modeling Software System Specifications," *IEEE Trans. Knowledge and Data Eng.*, pp. 238–252, vol. 4, June 1992.
[21] H. Graves, "Lockheed Environment for Automatic Programming," *IEEE Expert*, pp. 15–25, vol. 7, Dec. 1992.
[22] H. Graves, J. Louie, and T. Mullen, "A Code Synthesis Experiment," *Proc. Seventh Knowledge-Based Software Eng. Conf.*, pp. 6–17, McLean, Va., Sept. 1992.
[23] G.T. Heineman, G.E. Kaiser, N.S. Barghouti, and I.Z. Ben-Shaul, "Rule Chaining in Marvel," *IEEE Expert*, pp. 26–33, vol. 7, Dec. 1992.
[24] N.S. Barghouti and G.E. Kaiser, "Scaling Up Rule-Based Software Development Environments," *Int'l J. Software Eng. and Knowledge Eng.*, pp. 59–78, vol. 2, Mar. 1992.
[25] M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, and Y. Vassiliou, "Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis," *Proc. IEEE Int'l Symp. Requirements Eng.*, pp. 19–31, San Diego, Jan. 1992.
[26] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis, "Telos: Representing Knowledge About Information Systems," *ACM Trans. Information Systems*, pp. 325–362, vol. 8, Oct. 1990.
[27] M. Jarke, J. Mylopoulos, J.W. Schmidt, and Y. Vassiliou, "DAIDA: An Environment for Evolving Information Systems," *ACM Trans. Information Systems*, pp. 1–50, vol. 10, Jan. 1992.
[28] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach," *IEEE Trans. Software Eng.*, pp. 483–497, vol. 18, June 1992.
[29] B.A. Nixon, "Dealing with Performance Requirements During the Development of Information Systems," *Proc. IEEE Int'l Symp. Requirements Eng.*, pp. 42–49, San Diego, Jan. 1992.
[30] C. Rich and R.C. Waters, "Knowledge Intensive Software Engineering," *IEEE Trans. Knowledge and Data Eng.*, pp. 424–430, vol. 4, Oct. 1992.
[31] C. Rich and Y.A. Feldman, "Seven Layers of Knowledge Representation and Reasoning in Support of Software Development," *IEEE Trans. Software Eng.*, pp. 451–469, vol. 18, June 1992.
[32] B. Ramesh and V. Dhar, "Supporting Systems Development by Capturing Deliberations During Requirements Engineering," *IEEE Trans. Software Eng.*, pp. 498–510, vol. 18, June 1992.
[33] B. Ramesh and Luqi, "Process Knowledge Based Rapid Prototyping for Requirements Engineering," *Proc. IEEE Int'l Symp. Requirements Eng.*, pp. 248–255, San Diego, Jan. 1992.
[34] M.P. Heimdahl and N.G. Leveson, "Completeness and Consistency in Hierarchical State-Based Requirements," *IEEE Trans. Software Eng.*, pp. 363–377, vol. 22, June 1996.
[35] L.E. Moser et al., "A Graphical Environment for the Design of Concurrent Real-Time Systems," *ACM Trans. Software Eng. and Methodology*, pp. 31–79, vol. 6, Jan. 1997.
[36] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Trans. Software Eng. and Methodology*, pp. 231–261, vol. 5, July 1996.
[37] K. Miriyala and M.T. Harandi, "Automatic Derivation of Formal Software Specifications from Informal Descriptions," *IEEE Trans. Software Eng.*, pp. 1,126–1,142, vol. 17, Oct. 1991.
[38] M.T. Harandi and H.-Y. Lee, "Acquiring Design Schemas for Software Reuse," *Proc. Fifth Int'l Conf. Software Eng. and Knowledge Eng.*, pp. 491–498, Knowledge Systems Institute, 1993.
[39] B. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computer*, vol. 35, no. 8, 1986.
[40] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Proc. IEEE Symp. Logic in Computer Science*, pp. 428-439, 1990.
[41] R. Alur, C. Courcoubetis, and D. Dill. "Model-Checking in Dense Real-Time," *Information and Computation*, pp. 2–34, vol. 104, 1993.
[42] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "Partial-Order Reduction in Symbolic State Space Exploration," *Proc. Ninth Int'l Computer-Aided Verification Conf.*, pp. 340-351, Haifa, Israel, June 1997.

[43] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. "Symbolic Model Checking for Real-Time Systems," *Information and Computation*, vol. 111, no. 2, 1994.
[44] E.A. Emerson and A.P. Sistla, "Symmetry and Modelchecking," *Proc. Fifth Int'l Computer-Aided Verification Conf.*, Lecture Notes in Computer Science 697, pp. 463-478, Springer-Verlag, 1993.
[45] P. Godefroid and P. Wolper, "Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties," *Formal Methods in System Design*, pp. 149-164, Apr. 1993.
[46] G.J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.
[47] A. Valmari, "Stubborn Sets for Reduced State Space Generation," *Advances in Petri Nets*, pp. 463-478, Lecture Notes in Computer Science 483, Springer-Verlag, 1991.
[48] R. Milner, "A Calculus of Communicating Systems," Lecture Notes in Computer Science 92, 1980.
[49] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, Englewood Cliffs, N.J., 1985.
[50] J.A. Bergstra and J.W. Klop, "Algebra of Communicating Processes With Abstraction," *Theoretical Computer Science*, pp. 77-121, vol. 37, no. 1, 1985.
[51] G. Berthelot, "Checking Properties of Nets Using Transformations," Lecture Notes in Computer Science 222, pp. 19-40, 1986.
[52] M. Zhou, K. McDermott, and P.A. Patel, "Petri Net Synthesis and Analysis of a Flexible Manufacturing System Cell," *IEEE Trans. Systems, Man, and Cybernetics*, pp. 523-531, vol. 23, no. 2, Mar. 1993.
[53] R. Milner, "Operational and Algebraic Semantics of Concurrent Processes," *Handbook of Theoretical Computer Science*, J. van Leeuwen, ed., Elsevier, 1990.
[54] G. Boudol, G. Rucairol, and R. DeSimone, "Petri Nets and Algebraic Calculi of Processes," *Advances in Petri Nets*, Lecture Notes in Computer Science 188, 1985.
[55] G. Winskel, "Petri Nets, Algebras, Morphisms, and Compositionality," *Information and Computation*, pp. 197-238, vol. 72, 1987.
[56] K.K. Sabnani, A.M. Lapone, and M.U. Uyar, "An Algorithm Procedure for Checking Safety Properties of Protocols," *IEEE Trans. Comm.*, pp. 940-948, vol. 37, no. 9, 1989.
[57] S.M. Shatz, S. Tu, T. Murata, and S. Duri, "An Application of Petri Net Reduction for Ada Tasking Deadlock Analysis," *IEEE Trans. Parallel and Distributed Systems*, pp. 1,307-1,322, vol. 7, no. 12, Dec. 1996.
[58] A. Valmari, "Compositional State Space Generation," *Proc. 11th Int'l Conf. Application and Theory of Petri Nets*, pp. 43-62, 1990.
[59] E. Juan, J.J.P. Tsai, and T. Murata, "A New Compositional Method for Condensed State Space Verification," *Proc. First IEEE High-Assurance Systems Eng. Workshop*, pp. 104-111, Ontario, Canada, Oct. 1996.
[60] J.J.P. Tsai and E. Juan, "Efficient Compositional State-Space Verification for Communicating Processes in Distributed Systems," *Proc. Second IEEE High-Assurance Systems Eng. Workshop*, pp. 188-193, Washington, D.C., Aug. 1997.
[61] E. Juan, J.J.P. Tsai, and T. Murata "Compositional Verification of Concurrent Systems Using Petri-Net-Based Condensation Rules," *ACM Trans. Programming Languages and Systems*, vol. 20, no. 5, Sept. 1998.
[62] L. Aceto and M. Hennessy, "Termination, Deadlock, and Divergence," *J. ACM*, pp. 147-187, vol. 39, no. 1, Jan. 1992.
[63] P.C. Kanellakis and S.A. Smolka, "CCS Expressions, Finite State Processes, and Three Problems of Equivalence," *Information and Computation*, pp. 43-68, vol. 86, 1990.
[64] A. Valmari, "Compositional Analysis With Place-Bordered Subnets," *Proc. 15th Int'l Conf. Application and Theory of Petri Nets*, pp. 531-547, 1994.
[65] A. Valmari, "The Weakest Deadlock-Preserving Congruence," *Information Processing Letters*, pp. 341-346, vol. 53, 1995.
[66] M. Notomi and T. Murata, "Hierarchical Reachability Graph of Bounded Petri Nets for Concurrent-Software Analysis," *IEEE Trans. Software Eng.*, vol. 20, no. 5, May 1994.
[67] S. Duri, U. Buy, R. Devarapalli, and S.M. Shatz, "Application and Experimental Evaluation of State Space Reduction Methods for Deadlock Analysis in Ada," *ACM Trans. Software Eng. and Methodology*, pp. 340-380, vol. 3, no. 4, Oct. 1994.
[68] N.A. Lynch and M.R. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," *Proc. Sixth Symp. Principles of Distributed Computing*, pp. 137-151, ACM, New York, 1987.
[69] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J. M.M. Rutten, "On Blocks: Locality and Asynchronous Comm.," *Proc. Rex Workshop on Semantics: Foundations and Applications*, pp. 73-91, Lecture Notes in Computer Science 666, Springer-Verlag, 1993.
[70] T. Basten and M. Voorhoeve, "An Algebraic Semantics for Hierarchical P/T Nets," *Proc. 16th Int'l Conf. Application and Theory of Petri Nets*, pp. 45-65, Lecture Notes in Computer Science 935, 1995.
[71] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. Software Eng.*, Apr. 1995.
[72] D. Garlan, R. Allen, and J. Okerbloom, "Exploiting Style in Architectural Design Environments," *Proc. SIGSOFT: Foundations of Software Eng.*, ACM, Dec. 1994.
[73] N. Medvidovic, "A Classification and Comparison Framework for Software Architecture Description Languages," Technical Report No. UCI-ICS-97-02, Univ. of California, Irvine, Feb. 1997.
[74] J. Mostow, "Automating Program Speedup by Deciding What to Cache," *Proc. Ninth Int. Joint Conf. Artificial Intelligence*, vols. I-II, pp. 165-172, Los Altos, Calif., Morgan Kaufmann, 1985.
[75] O.V. Sokolsky and S.A. Smolka, "Incremental Model Checking in Modal mu-Calculus," *Proc. Sixth Int'l Computer-Aided Verification Conf.*, 1994.
[76] J.J.P. Tsai, A.P. Sistla, A. Sahay, and R. Paul, "Incremental Verification of Architecture Specification Language for Real-Time Systems," *Int'l J. Software Eng. and Knowledge Eng.*, vol. 8, no. 3, 1998.
[77] J.J.P. Tsai and K. Xu, "Specification of Multimedia Systems Using An Object-Oriented Architecture Description Language," *Proc. Sixth IEEE Int'l Conf. Multimedia Computing and Systems*, Florence, Italy, June 1999.
[78] J.J.P. Tsai and E. Juan, *Compositional Verification of High-Assurance Systems*, Kluwer, Boston, 1999.
[79] N. Medvidovic, P. Oreizy, J. E. Robbins, and R.N. Taylor, "Using Object-Oriented Typing to Support Architectural Design in the C2 Style," *Proc. SIGSOFT*, ACM, Oct. 1996.
[80] J. Magee and J. Kramer, "Dynamic Structure in Software Architectures," *Proc. SIGSOFT*, ACM, Oct. 1996.
[81] R. Allen and G. Garlan, "Formalizing Architectural Connection," *Proc. 16th Int'l Conf. Software Eng.*, pp. 71-80, May 1994.
[82] D.C. Luckham and J. Vera, "An Event-Based Architecture Definition Language," *IEEE Trans. Software Eng.*, pp. 717-734, Sept. 1995.
[83] S. Vestal, "MetaH Programmer's Manual," Version 1.09, technical report, Honeywell Technology Center, Apr. 1996.
[84] D. Garlan, R. Monroe, and D. Wile, "ACME: An Architecture Interchange Language," technical report, Carnegie Mellon Univ., Nov. 1995.
[85] P.C. Clements, "A Survey of Architecture Description Languages," *Proc. Eighth Int'l Workshop Software Specification and Design*, Germany, Mar. 1996.
[86] A. Abd-Alah and B. Boehm, "Models for Composing Heterogeneous Software Architectures," technical report, Univ. of Southern California, Los Angeles, 1996.
[87] M. Shaw and D. Garlan, "Characteristics for Higher-Level Languages for Software Architecture," technical report, Carnegie Mellon Univ., Dec. 1994.
[88] M. Shaw and D. Garlan, *Software Architecture: Perspectives On An Emerging Discipline*, Prentice Hall, 1996.
[89] J.J.P. Tsai, "Dependability of A.I. Systems," *IEEE Trans. Knowledge and Data Eng.*, pp. 57-63, vol. 6, no. 1, Feb. 1995.
[90] J.J.P. Tsai, D. Zhang, A. Sahay, and E. Juan, "Knowledge Verification," *Encyclopedia of Electrical and Electronics Eng.*, John Wiley and Sons, 1999.
[91] J.J.P. Tsai, B. Li, and E. Juan, "Parallel Evaluation of Software Architecture Specification," *Comm. ACM*, pp. 83-86, vol. 40, no. 1, Jan. 1997.
[92] J.J.P. Tsai, B. Li, *and* T. Weigert, "A Logic-Based Transformation Systems," *IEEE Trans. Knowledge and Data Eng.*, pp. 91-107, vol. 10, no. 1, Jan. 1998.
[93] J.J.P. Tsai and B. Li, *Knowledge-Based Software Architecture*, World Scientific Publisher, Singapore, 1999.

Jeffrey J.P. Tsai received his PhD degree in computer science from Northwestern University, Evanston, Illinois. He is a professor in the Department of Electrical Engineering and Computer Science at the University of Illinois at Chicago, where he is also director of the Distributed Real-Time Intelligent Systems Laboratory. He co-authored *Knowledge-Based Software Development for Real-Time Distributed Systems* (World Scientific, 1993), *Distributed Real-Time Systems: Monitoring, Visualization, Debugging, and Analysis* (John Wiley and Sons, 1996), and *Compositional Verification of High-Assurance Systems* (Kluwer, 1999); co-edited *Monitoring and Debugging Distributed Real-Time Systems* (IEEE Computer Society Press, 1995); and has published extensively in the areas of knowledge-based software engineering, expert systems, software architecture, computer-aided verification, high assurance systems, and distributed real-time systems. He received a University Scholar Award from the University of Illinois in 1994 and a Technical Achievement Award from the IEEE Computer Society in 1997. He is currently co-editor-in-chief of the *International Journal of Artificial Intelligence Tools*, and an editor of *IEEE Transactions on Knowledge and Data Engineering*, the *International Journal of Software Engineering and Knowledge Engineering*, and the *International Journal of Systems Integration*. He is a fellow of the IEEE and the AAAS.

Alan Liu received the BA degree in mathematics (with an emphasis in computer science) from San Jose State University in 1985, and the MS and PhD degrees in electrical engineering and computer science from the University of Illinois at Chicago in 1989 and 1994, respectively. He is currently an associate professor in the Department of Electrical Engineering at Chung-Cheng University, Taiwan. His research interests include knowledge-based systems, intelligent agents, and requirements engineering. He is a member of the IEEE, ACM, and TAAI.

Eric Juan received an MS degree in computer science from the Illinois Institute of Technology in 1993, and a PhD degree in computer science from the University of Illinois at Chicago in 1998. His current research interests include distributed and concurrent systems, real-time systems, and formal verification. He is a member of the IEEE and ACM.

Avinash Sahay received his BS degree in computer science from IIT, India, in 1996. He is currently a graduate student in the Department of Electrical Engineering and Computer Science at the University of Illinois at Chicago. His research interests include knowledge-based software engineering, computer-aided verification, and real-time systems.