Design methods for distributed software systems*

by CARL K. CHANG, MIKIO AOYAMA,** AND TSANG MING JIANG
University of Illinois at Chicago
Chicago, Illinois

ABSTRACT

As modern software systems tend to be more and more distributed, the design for such systems becomes very complicated. Design validation for distributed software systems is particularly difficult. Not only must a suitable design representation first be obtained, but automatic analysis tools have to be developed to validate essential design decisions and their impact on the resulting software. In other words, a formal specification method based on a chosen representation at the design level of distributed software systems is highly desirable.

This paper first reviews various design methods for distributed software systems. A new approach to design specification based on the well known Petri nets model is then presented. Methods for design validation of distributed software systems are also discussed.

*This work is currently supported by Fujitsu America, Inc. under contract number 052-80/136
**Aoyama is with Fujitsu Limited in JAPAN and currently visiting University of Illinois at Chicago.
INTRODUCTION

During the design phase, a computer system is normally specified as a model. Based on its model specification, a system can be gradually implemented. If the specification is formal, meaning that automatic analyses can be performed to a satisfactory extent, certain design errors can be detected and corrected at the design stage and later implementation errors can be avoided. Greater software economy can thus be achieved.

Specification of a computer system can be effectively used as a communication vehicle among all parties involved in the same development project, including both technical and non-technical personnel. Therefore, the system model should provide a precise and complete description of the modeled system so that desirable analyses can be carried out earlier and easier. Moreover, the model should be flexible enough to facilitate different views at different levels by all parties developing the same system.2,3,4

Distributed software systems, as opposed to conventional sequential software systems, become more and more popular. With the advent of fast advancing microprocessor technology as well as widely spread use of powerful workstations, software architects in various applications tend to adopt distributed computing architecture to implement highly effective and efficient software as solutions to a wide spectrum of real-world problems. In parallel to the adoption of distributed architecture, the design of a system to realize such an architecture becomes highly sophisticated.5

A variety of design methods have been proposed for real-world applications exhibiting distributed properties. The selection criteria used to determine design methods for the underlying models depend on the nature of the world, the design support environment tailored to the chosen model, and the analysis tools available in a particular environment.

This paper reviews two major classes of design methods. First, a number of representations based on the graph model are reviewed. Second, the one-dimensional specification language oriented design methods are surveyed. Pros and cons of different specification languages are discussed to address the suitability of applying them to distributed software design. The paper then proceeds to discuss design methodologies for distributed software systems from different perspectives pertaining to a number of distinguishable development paradigms. Design approaches of a specific application, namely, the switching software, are scrutinized to examine and determine the generality and usefulness of these methods.

A new design approach is then presented as a token of the new generation of design methods for distributed software systems. The approach integrates two forms of design representation: the graphical form and the language form. Advantages and applicability of this approach are then evaluated.

Finally, this paper addresses the design validation issue. In fact, design validation is very distinct, difficult, and important for distributed software systems. We point out future research directions in our conclusion.

DESIGN METHODS FOR DISTRIBUTED SOFTWARE SYSTEMS

Since the behavior of distributed software systems is very complicated, a desirable design method must be able to capture such behavior completely and translate it faithfully into a specification. For that purpose, a number of methods have been proposed. In general, they can be classified into two major groups: graph based and language based.

Graphical Approach

A number of graphical modeling methods have been developed.6 However, with the rapid advance of software technology, the architecture, requirements, and style of these graphical models have changed. Note that one of the original design representations is flow charts. The concept of structured programming introduces a number of standard structures into the topology of graphs. Meanwhile, hierarchical decomposition also has been playing a very important role in design modeling. Examples are SADT,7 HIPO,8 R_Net of SREM,9 and various other graphical models. Another driving force sprang from real-time software such as process control systems, switching systems, and communication protocols. Such systems require high performance and high reliability, hence precise descriptions of execution timing and process behavior are essential to modeling such systems. The State Transition Machine (STM) has been used widely for modeling real-time systems, especially switching systems.10,11 As a definite trend with the advancement of high-performance, systems are now evolving into (fully) distributed control. The introduction of distributed architecture requires significantly enhanced modeling capability due to the following reasons:

1. Complex behaviors such as concurrency, non-determinacy and asynchronism.
2. Needs of analytical capability, especially for real-time systems.

As we mentioned before, to model distributed software systems graphically, a number of modeling methods have been proposed. They can be classified into two groups: control-flow model and data-flow model. Control-flow models include Petri Nets (PN) and some extended PN models.12,13,14 The Data Flow Diagram (DFD)15 is an exam-
ple of the data-flow model. Since the original proposition by Petri in 1962, Petri nets have been the focus of many researchers because of not only their strong modeling capability, but also their various analytical capabilities based on graph theory. Important characteristics of Petri nets are summarized below:

1. Representation of concurrent execution of multiple processes.
2. Representation of non-determinant and asynchronous executions.
3. Modularity, that is, capability of various ways of decomposition and composition of multiple graphs.
4. Various analytical capabilities of model structuredness and dynamics.

However, Petri nets are a difficult tool to use to model systems involving timing requirements, data flow information, and stochastic information. Based on the generalized Petri nets, a number of extensions have been proposed as well as its subclasses. The relationship among these models has been investigated. However, to describe the behavior of distributed software systems comprehensively, it is necessary to represent both control flow and data flow. Yau and Caglayan proposed Modified Petri Nets (MPN) which integrated Petri nets and a data flow representation. MPN supports hierarchical decomposition as well as reuse of software components. Since PN is an asynchronous network model, it does not provide a timing concept. Recall that Ramchandani introduced Timed Petri Net (TPN). TPN is further extended as Stochastic Petri Net (SPN) and both models are applied to performance evaluation of distributed systems. We extended the MPN model by using a timer mechanism and defining more concrete data flow information. Our new model is referred to as Extended Modified Petri Net (EMPN) hereafter. A recent survey revealed some formalization effort on graphical models as visual languages. Moreover, the widely spread uses of graphical workstations make it easy to display graphical models and enhance the software development environments a great deal.

**Language Approach**

A good specification language should meet the following requirements.

1. What the system intends to do should be unambiguously described.
2. The specification written in that language should be complete for implementation.
3. Implementation errors should be precisely pinpointed based on the specification.

Because distributed software systems are more complicated than sequential ones, a specification language for distributed software systems, in addition to these basic requirements, should have features to accommodate the complexity of such systems. These features include formal development, easy human comprehension, and levels of abstraction.

Formal development refers to a set of well defined design processes to obtain the specification from informal and unstructured information. The Ina Jo specification language is an example as the formal specification language for the Formal Development Method (FDM). A specification language must be sufficiently precise so that a sequence of specification statements can be checked for consistency, non-redundancy, and completeness. This characteristic is normally in conflict with easy human comprehension. In fact, the diagrammatic representation is one way to improve human comprehension. As an example, in SREM, a graph representation is incorporated into the specifications to alleviate the problem.

In addition to these two features, to provide formal development facilities (usually through implementation refinement or function refinements), a distributed software system specification language must support levels of abstraction. For example, Event-Base Specification language (EBS) considers the conceptual models of distributed software systems from different levels of abstraction.

**Design Methodologies**

Much work has been done on actually designing software systems. However, the design methodology of real-time distributed software is not well developed so far. Designing such systems requires integrating more sophisticated techniques and procedures into the conventional methodology. Key issues to be considered in a well integrated design methodology include:

1. For real-time software,
   - rigorous timing design
   - performance design
   - quality design
2. For distributed software,
   - partitioning
   - allocation
   - communication design
3. For software productivity,
   - affirmity with modeling and testing methods
   - afficity with software support and automation

Due to space limitation, the authors concentrate on the design methodology for telecommunications software systems, such as switching software and communication protocols, which widely adopt the real-time distributed architecture. STM and its various extensions have been applied to the design of telecommunications software due to the fact that many telecommunications systems are actually finite state machines. However, significant enhancements to the current STM based design techniques are necessary because STM provides very limited design capabilities for such systems.

1. STM models force sequential thinking; concurrent computations are not expressed naturally.
2. Certain computations, such as those involving a queue or stack of arbitrary length, cannot be completely specified.
3. The performance cannot be evaluated directly.
4. The hierarchical structure or stepwise refinement are not supported naturally. Our model which will be discussed in the next section intends to provide some remedy to the limited specification power of most current design techniques especially for telecommunications software.

A NEW APPROACH

A basic model for distributed software systems is shown in Figure 1. An application world is normally divided into two sub-domains, namely, environment domain and system domain. The environment consists of many ports. Ports are abstract data types which are the sources of triggering stimuli to a system and the sinks of responses from the system. Port can be global or local. Each kind of port can be further divided into two different types. Active ports can generate signals on their own demand, while passive ports never generate signals unless on the system's demand. A system is composed of views that are executed in the course of functional components. As far as the interface is concerned, there are two levels of communications among views and components. At the view level, different views use the procedure calls to communicate. At the component level, components in different views use the message passing mechanism through ports for communication.

The design methodology based on this basic model is shown in Figure 2, in which each transition between successive steps is detailed as follows:

STEP 1 TO STEP 2: Each user's need is defined as a generic service.

STEP 2 TO STEP 3: Based on procedural requirements, find common procedures among all generic services. We define these common procedures collectively as a service view. The criteria that we may use in defining service views are:
1. number of processes
2. process size
3. hardware architecture
4. etc.

STEP 3 TO STEP 4: A functional component is the maximal set of functions under the same state which includes system markings and an environment for each service view.

STEP 4 TO STEP 5: Among all service views, find those common functional components, and define them as the kernel view.

The basic model can be represented by an EMPN which includes the data information (signals), timing requirements (timer ports), and the hierarchic structure (views). The EMPN can be described by a logic oriented language to be discussed in the next section.

EMPN DESCRIPTION LANGUAGE: EMPNDL

Based on the EMPN, the EMPNDL is developed not only to extend the model but to provide certain analysis capabilities without a graphic editor. EMPNDL is the description language of the EMPN model with the basic system model in mind. For example, the structure of a component that is
In our model, a system interfaces with its environment by exchanging signals with the environment. For example, in a telephone switch, lifting the handset is a caller’s signal to the system of the intention to dial a number. In response, the system sends a dial tone to the handset as a signal to the caller that dialing can now proceed.

Just identifying signals exchanged between the system and its environment is insufficient in describing the interface completely. For example, a telephone switching system involves two parties in its environment, namely, the caller and the callee. The two parties can be represented as two ports through which signals enter or leave a system.

By examining the nature of signals, two distinct signal types can be derived: impulse signals and level signals. An impulse signal is an infinitely short event, such as off-hook in the telephone switching system. An impulse signal can not only trigger system action but also carry data with it. For example, when the caller dials a number, the signal given to the switch is associated with a value, which is the number just dialed. On the other hand, a level signal represents a continuously monitored condition, such as the dial tone in the switching system, which lasts from the moment the caller goes off hook until the first digit of the number is dialed. A level signal may have several continuous conditions. As a real-world example, the colors of a traffic light signal have three levels: red, yellow, and green.

A signal can trigger a transition in the system. At some moment in time, the system is in a certain stable state, waiting for input. When the system receives a signal applicable to its current state, it performs a set of actions such as updating internal resources and sending messages to the environment. It then settles into another (not necessarily different) stable state, waiting for the next input.

Sometimes system’s behavior depends on whether an expected input signal from the environment arrives in a finite period of time. Consider the following requirement:

If the caller does not dial a number within 30 seconds, the switch shall send a howler tone to the caller.

To express this kind of time-dependent system behavior, the model provides the timer mechanism. A timer is a port which interprets signals as operations and responds properly. A timer can be an alarm clock, with a fixed timing interval. It starts running when it is explicitly started by the system. When the timer’s interval expires, the timer issues an alarm. An alarm has the same effect as an impulse signal in triggering a transition in the system. For example, to impose the above timing constraint upon a telephone switch, we can have a timer port called digit-receiving-timer. The system starts the digit-receiving-timer with 30 seconds when an off-hook signal from a caller is detected. If the caller dials a number promptly enough, the system will cancel the digit-receiving-timer. Otherwise, the digit-receiving-timer raises an alarm and triggers the system to send a howler tone to the caller.

In certain situations, system’s behavior may depend on the history of previous signals. Consider the following behavior of a telephone switch:

If the dialed number is inoperative, the user should receive a recorded message; if the dialed number is busy, the user should receive a busy tone.

The status of a dialed number is not part of the behavior of the telephone switching system to be specified. But, the telephone switch’s reaction to the status is and must be modeled. To handle this kind of hidden information, our model provides a decision construct. A decision is associated with a finite set of results. For example, the decision to be made for checking on the status of the dialed number may have three results: inoperative, busy, available. Depending on the result of a decision, a certain transition in the system may be triggered.

Our model provides the view mechanism to allow for the expression of the relationships among structural modules of a complex system. Systems in our model are specified mainly in terms of views. Each view is a subsystem that deals with a subset of the whole system’s signals. For example, a telephone switching system can have views of plain old telephone service (POTS), call forwarding service (CFS), automatic callback service (ACS). It is often the case that a specification is written by a team, rather than by a single author. The view mechanism lends itself very well to this situation, because views are independent components and can be elaborated separately. However, the combined views present a complete picture of the whole system. In spirit, views resemble modules in programming methodology and subschema in database theory.

**DESIGN VALIDATION**

Distributed software systems are more difficult to analyze than conventional centralized software systems. Distributed systems are inherently concurrent, asynchronous and nondeterministic. As an example, distributed switching systems can be verified by generating all reachable states and checking whether any of them is a nonprogressive state, such as deadlock, overflow, and unspecified receipt. This technique is referred to as state exploration. A major problem with state exploration is that it requires large execution time and storage. The problem is caused by the assumption that one needs to consider all possible progressive speeds for all parties contained in the distributed systems.

An efficient variation of state exploration for distributed switching systems has been studied. Briefly, in that study, the task of generating all reachable states is divided into N independent subtasks. In each subtask, only the states reachable by forcing maximal progress for one party are generated. Since the N subtasks are completely independent and, in most instances, the time and storage requirements for each
CONCLUSIONS

This paper reviews several software design techniques with most of the discussion centering around distributed software systems which are inherently nondeterministic, hard to design, and very difficult to analyze. A new model, EMPN, is presented for modeling distributed software systems. EMPN integrates representations of both control and data flow, and is extended primarily for real-time distributed software systems. The associated logic oriented language, EMPNDL, is also proposed for formal specification.

For further research, there are three directions to be investigated. First, there are many analysis tools available for PN models. However, due to the complexity of distributed software systems, the maximal progress technique should be utilized to reduce the complexity of analyzing the entire system.

Second, since the EMPNDL is a logic oriented language, it is natural to use AI reasoning techniques to find inconsistency, incompleteness, and redundancy at the specification level. As a tradition, these flaws are normally detected by a state transition matrix which expresses the relationships between the stimuli and responses based upon compiler techniques. An evaluation is necessary to compare these two techniques.

Finally, a testbed based on our new technique should be implemented to perform dynamic testing. Dynamic errors can thus be detected in this phase.

REFERENCES


