[4] F.T. Luk and H. Park, "Fault-tolerant matrix triangularization on systolic arrays," *IEEE Trans. Comput.*, vol. 37, pp. 1434–1438, Nov. 1988.
[5] J.Y. Jou and J.A. Abraham, "Fault-tolerant FFT networks," *IEEE Trans. Comput.*, vol. 37, pp. 548–561, May 1988.
[6] Y.-H. Choi and M. Malek, "A fault-tolerant FFT processor," *IEEE Trans. Comput.*, vol. 37, pp. 617–621, May 1988.
[7] D.L. Tao, C.R.P. Hartman, and Y.S. Chen, "A novel concurrent error detection scheme for FFT networks," in *Proc. 20th Int. Symp. Fault-Tolerant Comput.*, 1990, pp. 114–121.
[8] S.-J. Wang and N.K. Jha, "Algorithm-based fault tolerance for FFT networks," in *Proc. Int. Symp. on Circuits & Systems*, 1992, pp. 141–144.
[9] Y.-H. Choi and M. Malek, "A fault-tolerant systolic sorter," *IEEE Trans. Comput.*, vol. 37, pp. 621–624, May 1988.
[10] B. Vinnakota and N.K. Jha, "A dependence graph-based approach to the design of algorithm-based fault tolerant systems," in *Proc. Int. Symp. Fault Tolerant Comput.*, 1990, pp. 122–129.
[11] C.J. Anfinson and F.T. Luk, "A linear algebraic model of algorithm-based fault tolerance," *IEEE Trans. Comput.*, vol. 37, pp. 1599–1604, Dec. 1988.
[12] V.S.S. Nair and J.A. Abraham, "Real-number codes for fault-tolerant matrix operations on processor arrays," *IEEE Trans. Comput.*, vol. 39, pp. 426–435, Apr. 1990.
[13] J. Rexford and N.K. Jha, "Algorithm-based fault tolerance for floating-point operations in massively parallel systems," in *Proc. Int. Symp. on Circuits & Systems*, 1992, pp. 649–652.
[14] P. Banerjee and J.A. Abraham, "Bounds on algorithm-based fault tolerance in multiple processor systems," *IEEE Trans. Comput.*, vol. C-35, pp. 296–306, Apr. 1986.
[15] V.S.S. Nair and J.A. Abraham, "A model for the analysis of fault-tolerant signal processing architectures," in *Proc. 32nd Int. Tech. Symp. SPIE*, 1988, pp. 246–257.
[16] V.S.S. Nair and J.A. Abraham, "Probabilistic evaluation of on-line checks in fault-tolerant multiprocessor systems," *IEEE Trans. Comput.*, vol. 41, pp. 532–541, May 1992.
[17] D.J. Rosenkrantz and S.S. Ravi, "Improved upper bounds for algorithm-based fault tolerance," in *Proc. 26th Allerton Conf. Comm., Control and Computing*, 1988, pp. 388–397.
[18] D. Gu, D.J. Rosenkrantz, and S.S. Ravi, "Design and analysis of test schemes for algorithm-based fault tolerance," in *Proc. 20th Int. Symp. Fault-Tolerant Comput.*, 1990, pp. 106–113.
[19] V.S.S. Nair and J.A. Abraham, "A model for the analysis, design and comparison of fault-tolerant WSI architectures," in *Proc. Workshop on Wafer Scale Integration, Como, Italy*, June 1989.
[20] V.S.S. Nair and J.A. Abraham, "Hierarchical design and analysis of fault-tolerant multiprocessor systems using concurrent error detection," in *Proc. 20th Int. Symp. Fault-Tolerant Comput.*, 1990, pp. 130–137.
[21] R.K. Sitaraman and N.K. Jha, "Optimal design of checks for error detection and location in fault-tolerant multiprocessor systems," *IEEE Trans. Comput.*, vol. 42, pp. 780–793, July 1993.
[22] B. Vinnakota and N.K. Jha, "Diagnosability and diagnosis of algorithm-based fault-tolerant systems," *IEEE Trans. Comput.*, vol. 42, pp. 924–937, Aug. 1993.
[23] F.P. Preparata, G. Metze, and R.T. Chien, "On the connection assignment problem of diagnosable systems," *IEEE Trans. Electron. Comput.*, vol. EC-16, pp. 848–857, Dec. 1967.
[24] A.T. Dahbura and G.M. Masson, "An $O(n^{2.5})$-fault identification algorithm for diagnosable systems," *IEEE Trans. Comput.*, vol. C-33, pp. 486–492, June 1984.
[25] N.H. Vaidya and D.K. Pradhan, "System-level diagnosis: Combining detection and location," in *Proc. 21st Int. Symp. Fault-Tolerant Comput.*, 1991, pp. 488–495.

Using Petri Nets for the Design of Conversation Boundaries in Fault-Tolerant Software

Jie Wu and Eduardo B. Fernandez

***Abstract*—Only a few mechanisms have been proposed for the design of fault-tolerant software. One of these is the conversation, which, though it has some drawbacks, is a potentially promising structure. One of the problems with conversations is that they must be defined and verified by the user. In this short note, a systematic method for generating the boundaries of conversations directly from the specification is proposed. This method can also be used to verify conversations selected by the user. The specification is described by a high-level modified Petri net, which can easily be transformed into a state model called an action-ordered tree. The conversation boundaries are then determined from this tree. It is proved that the method proposed is complete in the sense that all of the possible boundaries can be determined, and it has the merit of simplicity. A robot arm control system is used to illustrate the idea. The proposed method can serve as the basis of a tool to assist in conversation designs.**

***Index Terms*— Concurrent software, conversation, fault-tolerant software, interprocess communication, Petri nets, software specification**

I. Introduction

As parallel processing takes in more and increasingly critical tasks such as air traffic control systems and mission critical systems, the reliability [18] of these systems becomes essential. Fault prevention and fault tolerance are two different strategies for achieving reliability. A fault-tolerant system is one that can continue to correctly perform its specified tasks in the presence of hardware or software faults. One way of achieving software fault tolerance is through redundant versions coupled with appropriate recovery. Two complementary approaches to recovery, known as *forward error recovery* and *backward error recovery* have been proposed [13]. Among the backward error recovery methods, the *conversation* scheme proposed by Randell [21] is one of the potentially promising methods for designing reliable concurrent software.

The conversation is a language construct that is a generalization to concurrent systems of the *recovery block* [21], used in sequential software, and defines an atomic action for a set of communicating processes. A conversation involves two or more processes and constitutes an atomic action enclosed by a set of boundaries. The boundaries of a conversation consist of a *recovery line*, *a test line*, and two *side walls*. A recovery line is a set of recovery points that are established before any process communication. A test line is the correlated set of the acceptance tests of the interacting processes. Processes within a conversation must be prevented from interacting with processes outside the conversation; that is, we must enforce two logical side walls. A conversation is successful only if all of the interacting processes pass their acceptance tests at the test line. If any of the acceptance tests fail, all of the processes within the conversation go back to the recovery line, recover to the previous state from the recovery points, and retry with their alternate try blocks.

The possible practical value of conversations was shown in a three-year experiment conducted at the University of Newcastle during

Manuscript received February 1993; revised February 20, 1994. This work was supported in part by the Florida High Technology and Industry Council.
The authors are with the Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431 USA.
IEEE Log Number 9403074.

1982–1984 [3]. A syntax and semantics to implement conversations in the context of programming language concepts were developed in [15] and [22]. Ways to improve conversation performance were discussed in [17] and [28]. A generalized architecture, able to support conversations (as well as other fault-tolerance constructs) was presented in [1], and some details of its implementation in an architecture using protection rings were described in [19]. Several variations of conversations were studied in [2] and [11]. A good survey of the possibilities and limitations of conversations was given in [9]. Further study of conversations is motivated because, together with the concept of *Programmer Transparent Coordination* [16], they are the only fault-tolerant mechanisms for concurrent processes, no convenient way to extend multiversion programming [5] to concurrent environments has been proposed. Although conversations have been studied extensively, there are still important practical problems that remain to be solved. One of these problems is the determination of the conversation boundaries.

The atomicity of the conversation makes it able to guarantee consistent recovery without exhibiting *domino effect* problems where a recovering process triggers a chain of process recoveries [21]. However, in a complex concurrent system, it is usually a difficult and tedious job for the user to define convenient boundaries for potential conversations. Tyrrell and Holding [24] proposed a method based on Petri nets that are used to describe system states. The dynamic behavior of the system was characterized by a state-change table derived from the state reachability tree. Functional attributes of the system states are then used to reduce the system state-change table to a table that includes only those states that are changed by interfunction actions. This method was later simplified in [26] by noting that these changes can occur only as the result of specific actions. These methods are applied to implementations of a concurrent system in some specific language and make use of one of the possible execution orders of the Petri net that describes the concurrent software. Therefore, these methods cannot generate all of the possible conversations.

We propose here a method to determine all of the conversation boundaries of an application directly from its specification. This method can also be used to verify the conversation boundaries selected by a user. The specification language used is a high-level modified Petri net. No features of the implementation language are shown in the specification; that is, the specification is general enough to allow different languages, especially different interprocess communication mechanisms, for implementation.[1]

Compared to the earlier methods, the proposed method has the following unique properties.

- **Generality:** The approach can assist the user in determining a potential conversation and to validate a conversation designed by the user.
- **Simplicity:** The approach uses a simple algorithm to determine or verify conversation boundaries.
- **Completeness:** This is the first approach that can systematically determine or verify all of the conversation boundaries.

Although the conversation design is a semantically driven process, we believe that the proposed method can be used to free the user from tedious design and verification details. In general, this method provides syntactic analysis while the user provides semantics analysis. We illustrate the proposed method as a definition tool in the text. Its use as a verification tool can be derived straightforwardly in the following way. Because the proposed method can determine all of the (correct) conversation boundaries, it can check whether a user-proposed conversation boundary is correct. If it is not, the generated conversations can be used as hints for the user to define better conversations.

This short note is organized as follows. First, we briefly describe Petri nets and their use as a specification language. A modified Petri net that shows only the process interactions is proposed to specify the concurrent software at a high level, and the state model of a Petri net is represented by an *action-ordered tree*. Using this tree, we develop a method to find all the possible conversation boundaries. The method is illustrated by applying it to the design of a robot arm controller. Finally, some conclusions are presented. More details and proofs can be found in [27].

II. Petri Nets as a Specification Language

There are numerous specification techniques for concurrent systems [8], [25], which can be either *model-oriented* or *property-oriented*. In the model-oriented approach, specification and design are explicit system models constructed out of abstract or concrete primitives. In the property-oriented approach, no explicit model is formulated; specifications are given in terms of axioms that define the relationships of operations to each other. Before selecting a specification technique, we should examine the features in the specification that are required to determine a conversation boundary.

To decide the conversation boundary, at least the following information is required:

1) the number of processes used (a process is a thread of control);
2) interprocess communication distribution (not necessarily the specific mechanisms for communication); and
3) execution order inside each process.

It is clear that property-oriented specification methods cannot be used to decide a conversation boundary, because they fail to provide the information listed above. Among the methods categorized as model-oriented, Petri nets [20] appear as a suitable specification language, although other models, such as the *UCLA Graph Model of Behavior* (GMB) [10], could be used. A Petri net is a bipartite directed graph. The two types of nodes in a Petri net are called *places* and *transitions*. Places are marked with tokens. A Petri net is characterized by an initial marking of places and a firing rule.

The concurrent software specification considered here can be organized hierarchically into several levels. We discuss here only the highest level of specification. In that level, only the necessary information required for conversation boundary determination, such as interactions among different processes, is shown. This specification can be represented by a modified Petri net, as described later. The actions inside each process are represented at a lower level of specification. The lower level of specification can be described by replacing each place in the specification with a more detailed Petri net through a refinement process. To facilitate comprehension, we concentrate on only the aspects that are related to determining conversation boundaries by showing only functional behavior of specification; nonfunctional behavior aspects [25], such as performance and timing [12], are not included.

In the highest level, we use a special notation to represent interprocess communication, because it plays an important role in the conversation boundary decision. The mechanization of interprocess communication should not be explicit in the specification. The only restriction to communication is that it be unidirectional and point-to-point. The method used here can easily be changed to accommodate the non-point-to-point cases. The Petri net specification of interprocess communication is shown in Fig. 1. Each of the sender

[1] To be more precise, the specification used here could be seen as a type of high-level implementation. Because the distinction between these two concepts is vague, we do not attempt to further distinguish them.

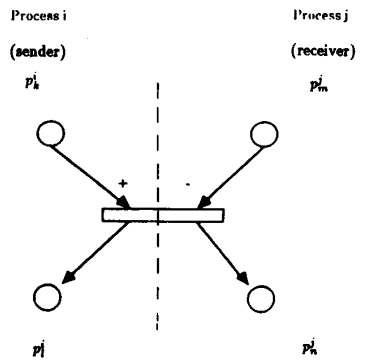


Fig. 1. Petri net specification of interprocess communication.

and receiver has a state before communication and a state after communication. States from different processes are separated by a dotted line. A thick bar is used to represent Petri nets with special firing rules in contrast with the thin bar used commonly in Petri nets. The firing rules depend on the implementation of interprocess communication, and there are three possible implementations [4]: synchronous, asynchronous, and buffered. The corresponding Petri net descriptions are shown in [27]. Our method of conversation boundary decision is independent of these implementations.

In the highest-level specification, we define for each process one place between every two interprocess communications. Two places are used to represent the initial and final state for each process. Another important feature of this Petri net specification (and also of the lower-level specifications) is that it is loop-free. To avoid loops in Petri nets, we need to examine three control structures: sequential, selective, and iterative. However, only iterative structures may cause loops. As discussed in [27], the way to avoid loops is to unfold them.

III. State Models of Petri Nets

We define each place in the Petri net specification graph as a state in a process. The state of the complete system, P_x, at a given time is defined as a vector $(p_x^1, \cdots, p_x^z)$, where p_x^y is the local state of process y in global state x $(1 \leq y \leq z)$, and z is the total number of processes. The local states of the processes over time define a partial order on the Petri net specification graph. For example, the partial order defined by the local states in Fig. 2 is $p_k^i < p_l^i$, $p_k^i < p_n^j$, $p_m^j < p_l^i$, $p_m^j < p_n^j$, $p_l^i = p_n^j$, and $p_k^i = p_m^j$, where "=" denotes two local states that have the same level. Repeated states (as those in unfolded loops) are not considered in the partial order.

With this definition, the system can be represented by a sequence of global process states as follows:

$$P_0 \xrightarrow{t_1} P_1 \xrightarrow{t_2} P_2 \cdots \text{ where } P_x < P_{x+1}.$$

If the following is true:

$$P_x = (p_x^1, p_x^2, \cdots, p_x^z) \text{ and } P_{x+1} = (p_{x+1}^1, p_{x+1}^2, ..., p_{x+1}^z),$$

then we have the following equation:

$$P_x < P_{x+1} \leftrightarrow \forall y \left[(p_x^y < p_{x+1}^y) \vee (p_x^y = p_{x+1}^y)\right] \wedge \exists y \left[p_x^y < p_{x+1}^y\right] \quad (1 \leq y \leq z),$$

Each t_i is an interprocess communication transition, and P_x is the global state before interprocess communication t_i. This sequence of global process states is called an *action-ordered tree.*

In an action-ordered tree, a state is derived from its "parent" state, either through a transition or through multiple transitions. For the case of multiple transitions, the new states are a summary of the possible states that can occur if only one transition fires at a time. The action-ordered tree is similar to the state reachability tree in [24], with the difference that only interprocess communications are involved. Therefore, the number of states used is drastically reduced in the action-ordered tree. The state-explosion problem, a typical problem in graph-based models, is thus alleviated.

IV. Conversation Boundary Decision

Conversation boundaries should be selected to ensure that the conversation is an atomic action; that is, a process participating in an atomic action (conversation) may exchange information only with other processes in that atomic action. In the case where several conversations are defined, the relationship between each pair of conversations should be either independence or proper nesting [21].

A. Test Line and Recovery Line Decision

The test line and the recovery line can be selected from the global states in the action-ordered tree. Note that the state for each process in the test line represents the state just after all interprocess communication has finished, whereas the state for each process in the recovery line represents the state just before any interprocess communication has happened. In order to find all of the possible test lines and recovery lines, we can view the lines selected from the action-ordered tree as a *virtual recovery line* and a *virtual test line.* The *actual recovery line* and *actual test line* can be easily derived by replacing each process state (local state) by a state in the subnet after a refinement of that local state. The refinement of a local state (a place in the Petri net) is performed by replacing that state by another subnet beginning and ending with a place. The actual state (place) is then chosen from among the totally ordered places in that subnet. By enumerating all of the possible refinements and selections of actual states for each refinement, we can get all of the possible test lines and recovery lines. In summary, the decisions for test lines and recovery lines usually require two steps, defined by Algorithm 1 below.

Algorithm 1:

1) Select virtual recovery and test lines (two ordered sets of states) from the action-ordered tree.
2) Select actual test and recovery lines from possible refinement subnets of each process state (local state) in the test and recovery lines.

Proposition 1: All of the possible recovery lines and test lines can be derived from Algorithm 1.

B. Side Wall Decision

As stated earlier, the side walls are defined by the specific set of process participating in the conversation. Two special cases should be considered, which we illustrate with the example of Fig. 2. In the first case, process PR_4 does not communicate with the other processes. Therefore, process PR_4 should not be included in the conversation. In terms of the action-ordered tree, this can be expressed as follows: These processes where the recovery state is the same as the test state should not be included in the conversation. For the second case, although PR_1 and PR_2, as well as PR_3 and PR_4, communicate with

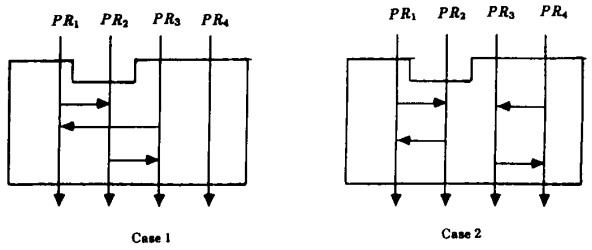


Fig. 2. Two special cases in deciding side walls.

each other, they do not communicate with the other pair. It is better here to build two conversations. In terms of action-ordered trees, we use the method of Algorithm 2 below.

Algorithm 2:

1) The virtual test line and the virtual recovery line are first determined.
2) Consider the states between the test line and the recovery line (include these two lines). If process PR_i communicates with process PR_j in these states, then we say that PR_i and PR_j have relation R, which can be written as $(PR_i, PR_j) \in R$. These pairs can be directly derived from the action-ordered tree.
3) Define R to be an equivalence relation (reflexive, symmetric, transitive).
4) Delete all the equivalence classes that consist of only one process, and each of the remaining classes will constitute a conversation.

C. Determination of the Boundaries of Multiple Conversations

For a pair of conversations C_i, C_j, denote their virtual recovery and test lines as P_{vr_i}, P_{vr_j} and P_{vt_i}, P_{vt_j}, respectively. The actual recovery lines are P_{ar_i}, P_{ar_j} and P_{at_i}, P_{at_j}, respectively. Case 1, including two subcases separated by conjunction ($\vee$), indicates that conversation C_i happens before conversation C_j. In Case 2, C_i is nested within C_j. Four subcases are identified in Case 2, Subcase 1 denotes that two conversations have no common virtual test or recovery lines. Subcase 2 indicates that two conversations share a common virtual recovery line, but no common actual recovery line. Subcase 3 is the same as Subcase 2 by interchanging the recovery and test lines. Subcase 4 represents two conversations that have common virtual test and recovery lines, but not common actual test or recovery lines. Two more cases can be obtained by exchanging the roles of i and j.

1) $(P_{vt_i} < P_{vt_j}) \vee [(P_{vr_i} = P_{vt_j}) \wedge (P_{at_i} < P_{ar_j})]$
2) $[(P_{vr_j} < P_{vr_i}) \wedge (P_{vt_i} < P_{vt_j})] \vee [(P_{vr_j} = P_{vr_i}) \wedge (P_{ar_j} < P_{ar_i}) \wedge (P_{vt_i} < P_{vt_j})] \vee [(P_{vr_j} < P_{vr_i}) \wedge (P_{vt_j} = P_{vt_i}) \wedge (P_{at_i} < P_{at_j})] \vee [(P_{vr_i} = P_{vr_j}) \wedge (P_{ar_j} < P_{ar_i}) \wedge (P_{vt_i} = P_{vt_j}) \wedge (P_{at_i} < P_{at_j})]$

Proposition 2: If there is a set of conversations where, for every pair of conversations, one of the two cases above is satisfied and the side walls for each conversation are decided by Algorithm 2, then these conversations are properly defined. That is, the two arbitrary conversations are either independent or properly nested.

The method to determine the boundaries for several conversations at the same time can be recursively defined as follows. Suppose that we have defined a set of conversations. One more conversation can be added to that set if every pair of conversations made up by the additional conversation and a conversation from the original set satisfies one of the two cases above.

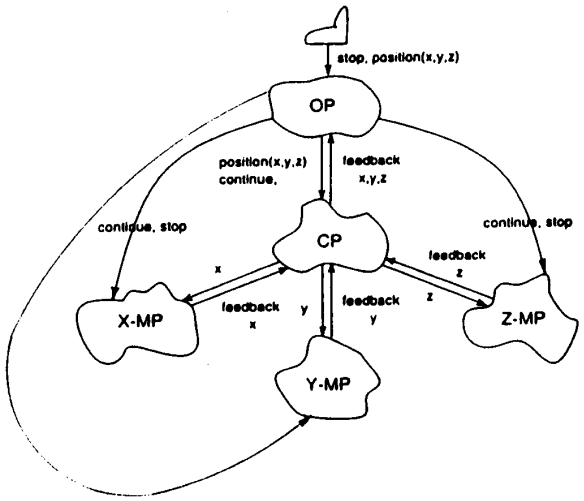


Fig. 3. Cooperating processes for robot controller.

D. Conversation Design

The proposed set of algorithms can be used in one of two ways:

1) as a definition tool to assist the designer to structure conversations from a given specification, or
2) as a verification tool to check the conversations selected by the designer.

In general, these tools should be used interactively, with the tool providing syntactic analysis and the designer providing semantic analysis. The use of the proposed method in conversation boundary decision can be described as follows.

1) The designer identifies those parts of the specification that have high criticality (semantic analysis).
2) The tool determines a set of conversations in the neighborhood of those parts (syntactic analysis).
3) The designer selects, based on certain criteria, a subset of the conversations offered by the tool. Criteria for selection could be, e.g., how easy it is to find appropriate acceptance tests, performance, and sections with possible catastrophic faults.

If multiple conversations are selected, these will be verified by the tool to ensure independence and proper nesting. This process will continue until a subset is selected that passes both syntactic and semantic analysis. Also, if necessary, the interaction should continue to refine the boundaries.

The use of the proposed method in verification could be constructed in a similar way. There may be several possibilities for conversations in an application. The user must select the most convenient conversation based on his knowledge of the application and the criticalness of each component of the application, and keeping in mind that large conversations reduce possible parallelism and have a deleterious effect on performance. Also, the feasibility of developing suitable acceptance tests must be considered. This process can be outlined as follows. The user selects a conversation based on his understanding of the application, and this method can then be used to build an action-ordered tree from the code to verify

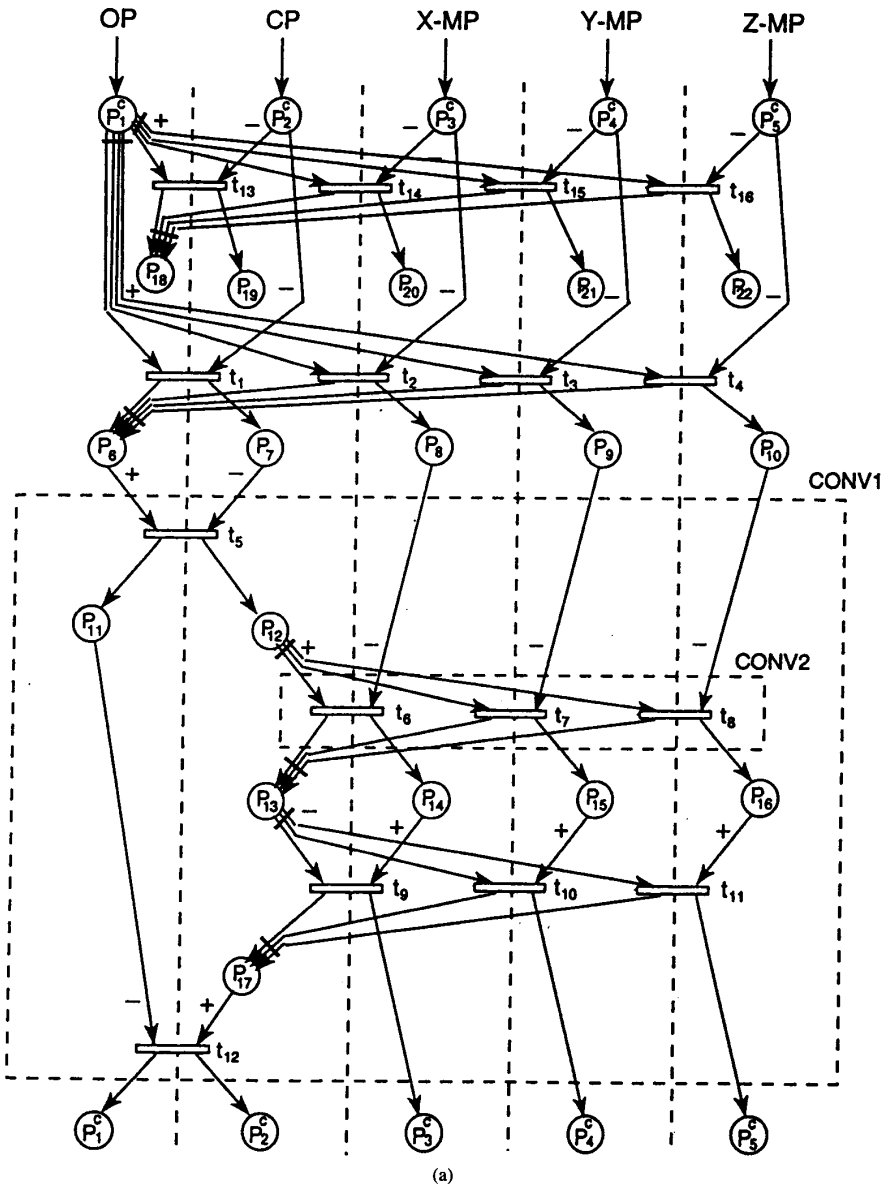


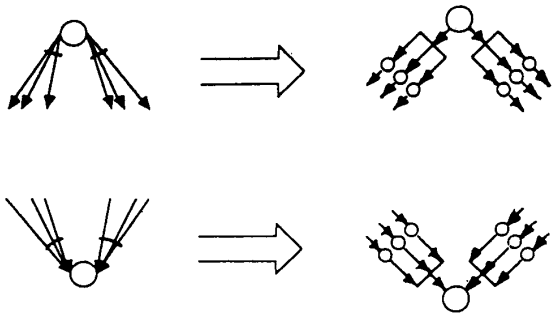
Fig. 4(a). Robot arm control specification.

that the selected conversation (or set of conversations) is a proper one.

V. Example: A Robot Arm Controller

We use here a robot arm controller to illustrate the method proposed above. Its specifications are similar to those described in [14]. The user inputs two types of commands: *position*, which orders the robot to move to a specific point (given by coordinates x, y, z), and *stop*, which stops every action. This system is designed to control a three-degree-of-freedom robot arm. A possible high-level specification can use five processes as follows. An operator process (OP), a control process (CP), and three motor processes for each x, y, z axis (X-MP, Y-MP, and Z-MP) (Fig. 3). The operator process receives the stop or position command from the keyboard. If it is a position command, this process also receives the coordinates of the desired position, and then transfers these values to the control process. The operator process also sends continue or stop signals to the motor processes. The control process calculates the relative directions and distances of the new coordinates, and then distributes these values to the three motor processes. The motor process will move the robot to the desired axial position. Fig. 4 shows a Petri net specification for the controller. A set of actions and a set of conditions are identified following the conventional Petri net modeling method. An action is represented by a transition and a condition (or state) is described by a place.

Fig. 4(b) describes a simplified notation used for Fig. 4(a), and the meaning of each state in Fig. 4(a) is shown in Table I. The



(b)

Fig. 4(b). The meaning of the simplified notation.

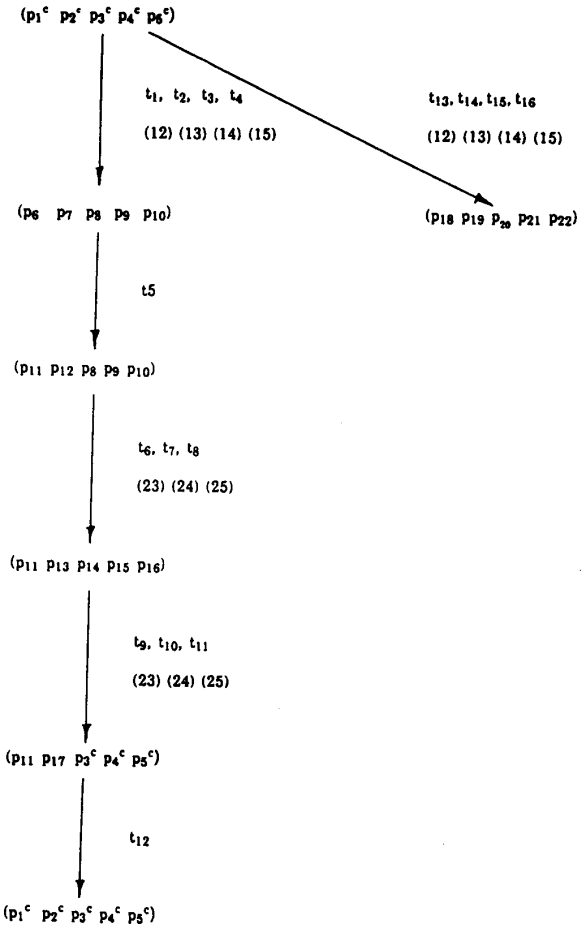
TABLE I
THE MEANING OF EACH STATE

	OP: operator process CP: control processMPs: X,Y,Z motor processes
p_1'	OP receives from the keyboard the stop or continue command and the command is sent to CP and the three MPs
p_2' p_3^c p_4^c p_5^c	initial states for CP and the three MPs respectively
p_6	the state for OP after CP and the three MPs all have received continue signals, and OP receives three new values for each axis, then OP passes these values to CP
p_7 p_8 p_9 p_{10}	the state for each CP, MP after receiving the continue signal from OP
p_{11}	OP state after sending the new values to CP
p_{12}	CP state after receiving the new values from OP, calculating the new coordinates, and then sending these values to the corresponding MP
p_{13}	CP state after sending the new values to MPs and then waiting for each motor to finish moving
p_{14} p_{15} p_{16}	each MP receives the new value from CP and then moves its motor
p_{17}	CP state after each MP finishes moving its motor and then signals to OP
p_{18}	OP state after CP and the three MPs receive the stop command and OP stops
p_{19} p_{20} p_{21} p_{22}	CP and each MP have received the stop command from OP and they stop

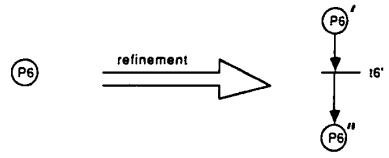
action-ordered tree for the robot arm example is shown in Fig. 5(a). This is obtained by summarizing the states after several transitions as described in Section III. In multiple transitions, each transition is associated with a pair of process ID's which indicate the two processes that participate in the communication. The numbers in parentheses indicate the processes associated with each communication. The superscript c in the first global state represents a loop condition. The second occurrence of the first global state is not really part of the partial order, it is used just to indicate a program loop.

When the proposed method is used as a definition tool, a possible scenario of user interaction is as follows.

1) The designer identifies the section that starts from receiving a new command at CP and ends at carrying out this command at each motor process as the critical one for this application.
2) The tool offers all the possible conversations, that is, from state $(p_6, p_7, p_8, p_9, p_{10})$ to state $(p_1^c, p_2^c, p_3^c, p_4^c, p_5^c)$ in the action-ordered tree. Since any combination of two states constitutes a recovery line and a test line for a conversation, and since there are five states (see Fig. 5(a)), a total of 10 potential conversations are offered.
3) A subset of conversations is selected by the designer where each conversation is represented by recovery and test lines. Suppose the *recovery line* and *test line* of two conversations, conv1 and conv2, are selected (Fig. 4(a)): $\{(p_6, p_7, p_8, p_9, p_{10}), (p_1^c, p_2^c, p_3^c, p_4^c, p_5^c)\}$ and: $\{(p_{11}, p_{12}, p_8, p_9, p_{10}), (p_{11}, p_{13}, p_{14}, p_{15}, p_{16})\}$, respectively.
4) Conversations conv1 and conv2 pass the feasibility check for multiple conversations and their boundaries (including side walls) could be completed as follows. By applying Algorithm 2 to conv1, only one equivalence class exists. Therefore, all of these processes should participate in this conversation. Because the operator process belongs to a class with only one element, it should not participate in conv2. If conv1 is denoted as C_i and conv2 is denoted as C_j, then C_i, C_j satisfies Case 4. Based on Proposition 2, these two conversations are properly defined. In fact, conv2 is nested within conv1 (as shown in Fig. 4(a)).



(a)



(b)

Fig. 5. (a) Action-ordered tree of robot arm control system. (b) Refinement of p_6 used in Fig. 4(a).

In conv1 if $(p_6, p_7, p_8, p_9, p_{10})$ is considered as a virtual recovery line, or a recovery state other than a state at interprocess communication is desired, then a necessary refinement can be applied. For example, a refinement of p_6 is shown in Fig. 5(b). In this figure, t_6' represents the action of receiving the command from the keyboard. State $_6$ can then be replaced either by states p_6' or p_6'' to get the actual local state. Note that in this case, another round of interaction is necessary.

VI. Conclusions

We have proposed a systematic way of determining or verifying conversation boundaries. A robot arm control system was used to illustrate the method. This approach not only has the advantage of being more complete and simpler than other methods but also has the merit of allowing a designer to make boundary decisions directly from specification. The specification used here is a high-level modified Petri net, which is organized in a hierarchical manner to narrow the cognitive gap between user and specifier. It is also a way to keep the specification concise and to show only the parts relevant to the boundary decision.

The proposed approach can also be used as a tool in an interactive software development environment. The user can specify which parts of the application he wishes to protect by conversations, and the environment would provide a set of well-formed conversations using the algorithms presented here. User selection would then be based on the issues mentioned above. As indicated above, the user could alternatively define the conversations, and the environment would verify that these are well-formed conversations.

Note that Petri nets are not the only tool that can be used to determine conversation boundaries. In general, the proposed method can be extended to be used by any model-oriented specification tool that can specify process communication and order of actions within a process, for example *timed Petri nets* [6] or *communicating real-time state machines* [23], to consider the effect on performance of the selected conversations, or the use of the GMB model, as suggested in [7].

Acknowledgment

We thank the anonymous referees for their valuable comments, suggestions, and corrections.

References

[1] M. Ancona, G. Dodero, V. Gianuzzi, A. Clematis, and E. B. Fernandez, "A system architecture for fault tolerance in concurrent software," *Comput.*, vol. 23, pp. 23–32, Oct. 1990.

[2] T. Anderson and J. C. Knight, "A framework for software fault tolerance in real time systems," *IEEE Trans. Software Eng.*, vol. 9, no. 3, pp. 355–364, May 1983.

[3] T. Anderson, P. A. Barrett, D. N. Halliwell, and M. R. Moulding, "Software fault tolerance: An evaluation," *IEEE Trans. Software Eng.*, vol. 11, no. 12, pp. 1502–1510, Dec. 1985.

[4] G. R. Andrews and F. B. Schneider, "Concepts and notations for concurrent programming," *Computing Surv.*, vol. 15, no. 1, pp. 3–43, Mar. 1983.

[5] A. Avizienis and J. P. J. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Comput.*, vol. 17, no. 8, pp. 67–80, Aug. 1984.

[6] B. Berthomieu and M. Diaz, "Modeling and verification of time dependent systems using timed Petri nets," *IEEE Trans. Software Eng.*, vol. 17, pp. 259–273, Mar. 1991.

[7] G. F. Carpenter and A. M. Tyrrell, "The use of GMB in the design of robust software for distributed systems," *Software Eng. J.*, pp. 268–282, Sept. 1989.

[8] B. Cohen, *The Specification of Complex Systems.* Reading, MA: Addison-Wesley, 1986.

[9] F. Di Giandomenico and L. Strigini, "Implementations and extensions of the conversation concept," *Proc. 5th GI/ITG/GMA Int. Conf. Fault-Tolerant Comput. Syst.*, Nürnberg, Germany, 1991.

[10] G. Estrin, R. S. Fenchel, R. R. Razouk, and M. K. Vernon, "SARA, System architect's apprentice: Modeling, analysis, and simulation support for design of concurrent systems," *IEEE Trans. Software Eng.*, vol. 12, no. 2, pp. 293–311, Dec. 1986.

[11] S. T. Gregory and J. C. Knight, "A new linguistic approach to backward error recovery," *Proc. 15th Int. Symp. Fault Tolerant Comput.*, 1985, pp. 404–409.

[12] H. Hecht and M. Hecht, "Fault tolerant software," in D.K. Pradhan, Ed., *Fault Tolerant Computing: Theory and Techniques.* Englewood Cliffs, NJ: Prentice-Hall, 1986, pp. 658–696.

[13] P. Jalote and R. H. Campbell, "Atomic actions for fault tolerance using CSP," *IEEE Trans. Software Eng.*, vol. 12, no. 1, pp. 59–68, Jan. 1986.

[14] J. M. Kerridge and D. Simpson, "Three solutions for a robot arm controller using Pascal-Plus, Occam, and Edison," *Software Practice and Experience*, vol. 14, no. 1, pp. 3–15, Jan. 1984.

[15] K. H. Kim, "Approaches to mechanization of the conversation scheme based on monitors," *IEEE Trans. Software Eng.*, vol. 8, pp. 189–197, May 1982.

[16] ——, "Programmer-transparent coordination of recovering concurrent processes: Philosophy and rules for efficient implementation," *IEEE Trans. Software Eng.*, vol. 14, pp. 810–821, June 1988.

[17] K. H. Kim and S. M. Yang, "Performance impact of look-ahead execution on the conversation scheme," *IEEE Trans. Comput.*, vol. 38, pp. 1188–1202, Aug. 1989.

[18] J. C. Laprie, "Dependability: A unifying concept for reliable computing an fault tolerance," in T. Anderson, Ed., *Dependability of Resilient Computers.* New York: BSP, 1989, pp. 1–28.

[19] B. M. Ozaki, E. B. Fernandez, and E. Gudes, "Software fault tolerance in architectures with hierarchical protection levels," *IEEE Micro*, vol. 8, pp. 30–43, Aug. 1988.

[20] J. L. Peterson, *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[21] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. 1, no. 2, pp. 220–232, June 1975.

[22] D. L. Russell and M. I. Tiedeman, "Multiprocess recovery using conversations," *Proc. 9th Int. Conf. Fault-Tolerant Comput.*, 1979, pp. 106–109.

[23] A. C. Shaw, "Communicating real-time state machines," *IEEE Trans. Software Eng.*, vol. 18, no. 9, pp. 805–816, Sept. 1982.

[24] A. H. Tyrrell and D. J. Holding, "Design of reliable software in distributed systems using the conversation scheme," *IEEE Trans. Software Eng.*, vol. 12, no. 9, pp. 921–928, Sept. 1986.

[25] J. M. Wing, "A specifier's introduction to formal methods," *Comput.*, vol. 23, pp. 8–24, Sept. 1990.

[26] J. Wu and E. B. Fernandez, "A simplification of a conversation design scheme using Petri nets," *IEEE Trans. Software Eng.*, vol. 15, pp. 658–660, May 1989.

[27] ——, "Using Petri nets for fault tolerance in concurrent software," Tech. Rep. TR-CE-90-4, Dept. of Comput. Sci. and Eng., Florida Atlantic Univ., Boca Raton, FL, USA, 1990.

[28] S. M. Yang and K. H. Kim, "Implementation of the conversation scheme in message-based distributed computer systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, pp. 555–572, Sept. 1992.