White Rose Research Online
eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of York.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/971/

Article:

Tyrrell, A.M. and Carpenter, G.F. (1995) CSP methods for identifying atomic actions in the design of fault tolerant concurrent systems. IEEE Transactions on Software Engineering. pp. 629-639. ISSN: 0098-5589

https://doi.org/10.1109/32.392983

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UNIVERSITY OF LEEDS
University of Sheffield
UNIVERSITY of York

CSP Methods for Identifying Atomic Actions in the Design of Fault Tolerant Concurrent Systems

Andrew M. Tyrrell, *Member, IEEE*, and Geof F. Carpenter

***Abstract*—Limiting the extent of error propagation when faults occur and localizing the subsequent error recovery are common concerns in the design of fault tolerant parallel processing systems. Both activities are made easier if the designer associates fault tolerance mechanisms with the underlying atomic actions of the system. With this in mind, this paper has investigated two methods for the identification of atomic actions in parallel processing systems described using CSP. Explicit trace evaluation forms the basis of the first algorithm, which enables a designer to analyze interprocess communications and thereby locate atomic action boundaries in a hierarchical fashion. The second method takes CSP descriptions of the parallel processes and uses structural arguments to infer the atomic action boundaries. This method avoids the difficulties involved with producing full trace sets, but does incur the penalty of a more complex algorithm.**

***Index Terms*—Atomic actions, concurrent systems, CSP, fault tolerance.**

I. INTRODUCTION

A DISTRIBUTED processing system, comprising a set of discrete processing units, offers the user not only the prospect of increased efficiency and throughput through parallelism, but its inherent redundancy might also be exploited to enhance reliability. To do so requires a properly designed fault tolerance infrastructure which maintains the integrity of the system under fault conditions. This paper describes CSP-based methods which facilitate the placement of fault tolerance software structures across a distributed system to ensure safe operations in the presence of faults.

Notwithstanding the use of standards and guidelines [1], [2], [3] in the design of software-based real-time systems for safety-critical applications, and the concomitant adoption of formal methods, it is probable that faults will still be introduced into a design either explicitly as part of a particular component or implicitly through the omission of a particular feature. It is unrealistic to expect all software design faults to be detected during design and testing, and latent faults may persist into system use [4].

Fault tolerance [5] is often incorporated into a design as a ruggedization process to protect a process or set of processes regarded as critical to safe system operation. The fault tolerance mechanisms are required to recognize faults by the errors they cause and to prevent error migration from the faulty process to elsewhere in the system, so that error recovery is localized. The extent of the error recovery operation can be limited if a boundary can be identified within the state-space of the distributed system across which error propagation by interprocess communication is impossible; it must include all processes which interact with the function being protected and exclude all processes that do not interact with it. In other words, the state-space of the system has to be partitioned into a hierarchy of atomic actions [6]. It is then possible to introduce a distributed error detection and recovery mechanism around the atomic action [7] which ensures that all the processes affected by the fault cooperate in recovery. This localization of fault tolerance simplifies the design and can help to meet timing constraints in real-time systems [8].

Methods for determining hierarchical sets of atomic actions are not widely known. This paper describes methods which use the mathematically based notation of Communicating Sequential Processes (CSP) [9] to describe the operation of a distributed system, and the interactions between the processes. The analysis allows the designer to identify hierarchical sets of atomic actions within the design. The model of the system can then be used to place fault tolerance software structures, correctly including all participants.

II. ATOMIC ACTIONS AND FAULT-TOLERANCE

To an external observer the activity of a process is defined by its sequence of external interactions; any internal actions (of which there may be many) can not affect the external observer, at least until the next external interaction. This allows the concept of an atomic action to be derived [6]: the activity of a set of processes is defined as an atomic action if there are no interactions between that set of processes and the rest of the system for the duration of that activity. The extension to hierarchically nested atomic actions is straightforward. These concepts are well-known in distributed transaction processing [10] from which field many other attributes of atomic actions, such as serializability, failure atomicity and permanence of effect can be defined.

The process of identifying the atomic actions within a parallel system design brings into clear focus the structure of interprocess interactions and thus the route by which errors might propagate under fault conditions. All common mechanisms for providing fault tolerance in parallel systems, such as forward error recovery [11], N-version programming [12], conversations [11], consensus recovery blocks [13] and distributed recovery blocks [14], have to cope with error confinement and achieve this by imposing logic structures "around" atomic actions [15].

Manuscript received June 1993; revised August 1994.
A.M. Tyrrell is with the Department of Electronics, University of York, Heslington, York, Y01 5DD, UK; e-mail: amt@ohm.york.ac.uk.
G.F. Carpenter is with the Department of Electrical and Electronic Engineering and Applied Physics, Aston University, Aston Triangle, Birmingham, B4 7ET, UK.
IEEECS Log Number S95020.

0098-5589/95$04.00 © 1995 IEEE

A generalized fault tolerant mechanism could be considered as a coordinated set of recoverable blocks, with one recoverable block in each interacting process, allowing distributed error detection and recovery. The mechanism is bounded by an entry line, an exit line and two side walls which completely enclose the set of interacting processes which are party to the mechanism, and across which interprocess interactions are prohibited. The structure is indicated diagramatically in Fig. 1.

The entry line defines the start of the atomic action and consists of a coordinated set of recovery points for the participating processes. The exit line comprises a coordinated set of acceptability tests. Only if all participating processes pass their respective acceptability tests is the mechanism deemed successful and all processes exit, in synchronism, from the action. If any acceptability test is failed, recovery is initiated and processing "passed" to another set of recoverable processes. Thus all processes in the atomic action cooperate in error detection.

The duality of atomic actions and recovery mechanisms has been discussed at length in [10]. Atomic actions can be viewed as modeling an "object-action" type of system where atomic actions operate on objects. Expressed graphically as an action diagram (Fig. 2) circles represent actions, and arcs show the dependencies between actions. Thus, in Fig. 2, action A_2 uses objects "x" and "y" released by action A_1. Similarly, action A_4 uses "y" when it has been released by action A_2. A comparison with Fig. 1 shows that the recovery mechanism is the dual of the action and the process is the dual of the object; a mechanism C_i is replaced by action A_i with an arc connecting A_i with A_j if C_i and C_j have processes in common. Thus, Fig. 2b and Fig. 2c can be regarded as duals. In the context of this paper, for example, action A_3 provides a fault tolerant function operating on processes P, Q, and R.

Any attempt to incorporate an entry line and an exit line at arbitrary locations in a concurrent system is unlikely to lead to a properly formed recovery mechanism. It is necessary to identify a boundary within the state space of the complete set of processes across which error propagation by communication is prevented. Clearly, this boundary will be the boundary of an atomic action, since such a boundary of necessity prohibits the passing of information to any process not involved in the atomic action and similarly embraces all interacting processes within the atomic action. Recovery mechanisms can be nested systematically in the same hierarchical fashion as atomic actions. If this duality is not imposed, then should the system attempt to backtrack and recover in response to a fault, progressive collapse by the domino effect [11] can occur.

In the literature, strategies for implementing fault tolerance in parallel systems [16], [17], [18], [19] and for handling problems which occur if the chosen mechanism is incorrectly located, have received more detailed attention than the fundamental problem of placing the mechanisms correctly. Correctly placed mechanisms, coincident with atomic action boundaries, avoid error propagation problems. This paper is concerned with the analysis of a prototype design for atomic actions. Ideally, a design method would incorporate the requisite, appropriately placed, atomic actions and the associated fault tolerance infrastructure into a system with a minimal amount of reanalysis and redesign, and an eventual goal is to define such a design method. However, the techniques are still insufficiently mature for this to be achieved and consequently this paper retains the normal design practice in which fault tolerance mechanisms are superimposed upon selected atomic actions and the new designs subjected to reanalysis.

III. State Space Methods for Identifying Atomic Actions

Substantial work has been performed on the ability to model systems, and to reason about their behavior, using state space representations such as Petri nets or GMB [20], [21]. In the Petri net approach, each process state can be associated with a Petri net place, and each state transition with a Petri net transition [22]. Process execution is simulated by allowing marking tokens to flow through the Petri net. From the formulation of a reachability graph, the behavior of the Petri net, and therefore of the modeled system, can be analyzed.

Experience with occam [23] as a design language for loosely-coupled real-time concurrent systems [24], [25] has led to Petri net methods for identifying atomic actions. By only permitting synchronous, atomic, communications, occam forces communicating processes into mutual synchronization at communication points. This not only imposes a strict discipline on the designer (because errors in the synchronization logic can lead to deadlock) but also leads to a system more

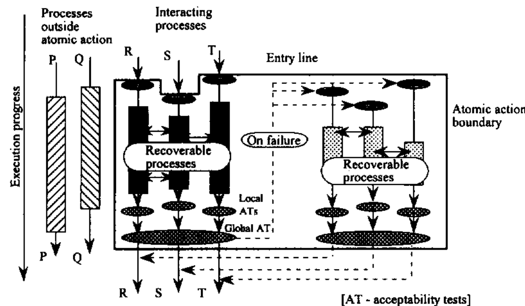


Fig. 1. The structure of a fault tolerant mechanism involving processes R, S, and T.

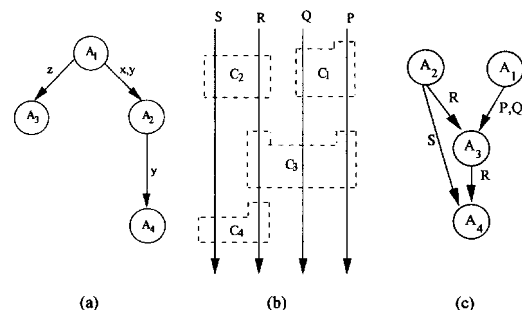


Fig. 2. (a) an action diagram; (b) a process-recovery diagram; (c) the action diagram dual of Fig. 2b.

amenable to analysis. The system is designed using the requirement specification and modeled as a Petri net. Examination of the state reachability graph permits the designer to identify the boundaries of atomic actions. Inspection determines which atomic action boundary encloses which system function, and an appropriate error detection and recovery mechanism to protect any chosen system function can then be incorporated at the level of the atomic action without disturbing the constituent processes or their interprocess actions.

Although the method is effective, it requires:

1) translation of an existing textual occam design into a graphical Petri net;
2) translation between the graphical Petri net and set theory or matrix-based methods for reachability analysis;
3) translation of the identified atomic action entry and exit points back to the original occam design;

which are made more difficult because:

4) for all but the simplest examples, there is a computational explosion which could restrict the analysis.

Although automated tools exist for these translation processes, often error-prone manual methods are still involved. For Petri-net-based methods the designer must be satisfied that the translation steps 1–3 do not themselves introduce errors.

Occam has a mathematical basis in the theory of Communicating Sequential Processes (CSP) [9]. CSP permits a fundamental description of a concurrent processing system in terms of the component processes, the interactions between the processes, and interactions with the real-world environment. Since a CSP description is directly amenable to mathematical analysis, it is possible to decide behavioral properties, such as the presence of reachability pathologies, without the need for error-prone translation into a complementary representation. The ability to reason about timeliness in recent extensions to CSP [26] should further promote its use in the design of time-critical and safety critical systems.

The trace of a CSP process is a record of the sequence of events in which a process could engage and indicates directly a possible execution behavior of that process [27]. During the design phase it would be advantageous to determine all the possible traces which a process might produce. This procedure is termed trace evaluation in this paper. For any but the simplest process there will be a number of possible traces; for a set of concurrently executing processes the overall trace set will be all permitted interleavings of the traces of the component processes. If the processes interact only by synchronous communications, then the processes are brought into synchronism for the communication event. The communication event will be in the alphabet of both the communicating processes and will constrain the set of all possible traces.

It is not practicable to create the complete set of traces unless the set of processes is subject to certain constraints:

1) The processes must terminate, or arrive at a previously reached state, in a finite number of steps, else the set of traces becomes infinite.
2) Where program flow is made dependent on the value of variable expressions, static analysis has to consider all possible values within the range of the variable expression, which may be infinite and lead to an infinite set of traces.
3) No. 2) precludes from analysis classes of loops where trace evaluation would have to evaluate loop guards, and also the use of subscripted communication channels where the subscript is determined by a variable expression.
4) Guarded choice (and thus nondeterminism) can be included provided the truth value of the guard is reflected in the trace set.
5) Interprocess communications occurring in loop constructs pose major problems for trace evaluation; in particularly if the loop iteration is controlled by a variable expression which is even indirectly determined by the real-world environment, then analysis can only be performed for special cases, i.e., where a subset of these environmental values are considered.
6) Certain commonly occurring forms of loop can be handled; for example, if the loop is executed a predefined number of times (e.g., the conventional FOR loop) and the number of communications in both processes exactly match, or if both communicating processes have matched loops which iterate synchronously in both processes (as in the real world robot example).

Trace evaluation can be tedious and error-prone if performed manually, but it may be readily automated. An automated tool, termed CoPla, has been built at the University of York within an X-Windows environment [28].

IV. CSP and Atomic Action Identification

Trace analysis can be used to identify atomic actions within a CSP design and to infer a hierarchical arrangement of these atomic actions. The technique presented here is inspired by the successful Petri net methods [29]; it requires the designer to evaluate all the possible execution traces for the CSP design and then to analyze process execution for events which are interprocess communications. By definition, the activity of a set of processes constituting an atomic action is such that no interactions take place between that set of processes and the rest of the system. Consequently the boundary of the atomic action can then be used for the proper incorporation of coordinated error detection and error recovery mechanisms within CSP designs.

Conventionally the complete set of possible traces for a process, P, is designated by

$$\text{traces}(P) = \{t_1, t_2, \ldots, t_k\}$$

where

$$t_i = \langle e_{i1}, e_{i2}, \ldots, e_{ij}, \ldots, \surd \rangle$$

and the event e_{ij} corresponds to the jth event in the ith possible trace t_i. $\surd$ is the successful termination event. (Strictly speaking, $\langle e_{i1} \rangle$, $\langle e_{i1}, e_{i2} \rangle$, and all intermediary event sequences are also members of traces(P) as well as $\langle e_{i1}, e_{i2}, \ldots, e_{ij}, \ldots, \surd \rangle$; this paper only consider traces with the termination event.)

The algorithm for trace evaluation is a straightforward application of continuous simplification. Given P, all the events

e_{i1} which can be the first element of the trace are extracted, to yield a simpler process P/e_{i1} (P after engaging in e_{i1}), thus:

$$\text{traces}(P) = \text{traces}(e_{11} \wedge (P/e_{11})) \cup \text{traces}(e_{21} \wedge (P/e_{21})) \cup \ldots$$
$$= \bigcup_{i=1}^{n} \text{traces}(e_{i1} \wedge (P/e_{i1}))$$

where $\wedge$ is the catenation operator. The function $\text{traces}(e_{i1} \wedge (P/e_{i2}))$ can then be evaluated in a similar fashion.

Consider N processes in concurrent execution:

$$\mathcal{P} = P_1 \parallel \ldots \parallel P_n \parallel \ldots \parallel P_N$$

As before, for each component process, P_n:

$$\text{traces}(P_n) = \{t_1, t_2, \ldots, t_k\}$$

where

$$t_i = < e^n_{i1}, e^n_{i2}, \ldots, e^n_{ij}, \ldots, \surd^n >$$

Note, a superscript character is added to show that e^n_{ij} and $\surd^n$ occur within process P_n. Each event may be further categorized, either as being local to its constituent process (thus, l^n_s, appearing only in the alphabet of process, P_n) or as being a communication event (thus, c^{nm}_s, appearing in the alphabets of both processes P_n, P_m, which participate in the communication, and thereby forcing synchronization). For each process P_n, the local events l^n_s form the set L_n and the communication events c^n_s form the set C_n; thus:

$$\alpha(P_n) = L_n \cup C_n$$
$$l^n_s \in L_n$$
$$c^n_s \in C_n$$

The traces of the set of processes $\mathcal{P}$ will be all permitted interleavings of the traces of the component processes, written as:

$$\text{traces}(\mathcal{P}) = \{t_1, t_2, \ldots, t_k\}$$

where

$$t_i = < g_{i1}, g_{i2}, \ldots, g_{ij}, \ldots, \surd >$$

Here, the event g_{ij} corresponds to the jth event in the ith possible trace t_i of traces($\mathcal{P}$). This general event g_{ij} is either an element from the alphabet of one of the constituent processes if it is a local event; otherwise it must appear in the alphabet of exactly two processes as a communication event. Thus:

$$\exists n: g_{ij} \in \alpha(P_n)$$
$$(g_{ij} \in L_n) \vee (g_{ij} \in C_n \wedge g_{ij} \in C_m \wedge n \neq m)$$

The method for identifying hierarchically nested atomic actions is defined in algorithms 1 and 2. Algorithm 1 defines how the entry and exit lines to the atomic action are identified.

A. Algorithm 1

Given three processes P_p, P_q, P_r in parallel execution (with obvious extension to more than three processes):

1) Add before the start of each process the special events: e^p_{init}, e^q_{init}, and e^r_{init}; recall that the last event in each process is followed by $\surd^p$, $\surd^q$, and $\surd^r$, respectively.
2) Select a sequence of consecutive events
$$e^p_{s1} \to \ldots \to e^p_{sj} \to \ldots \to e^p_{sn}$$
within P_p which are to be constituents of the atomic action. The sequence must enclose fully any parallel or selection constructs within the sequence. Note that e^p_{init} and $\surd^p$ will not be part of this sequence.
3) Define the empty sets S, F, K, J.
$$S := \{\}; F := \{\}; K := \{\}; J := \{\}$$
4) Generate traces($\mathcal{P}$), (including e^p_{init} and $\surd^p$).
5) For each trace t_i in traces($\mathcal{P}$), locate $g_{im} = e^p_{s1}$. Add $g_{i(m-1)}$ to set S.
$$(\forall t_i) | (t_i \in \text{traces}(\mathcal{P}))$$
$$(\forall g_{im}) | (g_{im} \in t_i) \wedge (g_{im} = e^p_{s1}) \to S := S \cup g_{i(m-1)}$$
6) For each trace t_i in traces($\mathcal{P}$), locate $g_{in} = e^p_{sn}$. Add $g_{i(n+1)}$ to set F.
$$(\forall t_i) | (t_i \in \text{traces}(\mathcal{P}))$$
$$(\forall g_{in}) | (g_{in} \in t_i) \wedge (g_{in} = e^p_{sn}) \to F := F \cup g_{i(n+1)}$$
7) Compute the set difference K = S–F. This defines the complete set of events which must immediately precede the start of the atomic action.
8) Compute the set difference J = F–S. This defines the complete set of events which must immediately follow the end of the atomic action.

B. Justification of Algorithm 1

Initially, before algorithm 1 is executed:

$$K = \{\}, \; J = \{\}, \; S = \{\}, \; F = \{\}$$

The sequence of events in P_p which are to be constituents of the atomic action are described as:

$$e^p_{s1} \to \ldots \to e^p_{sj} \to \ldots \to e^p_{sn}$$

If all $e^p_{sj} \in L_p$ then no interprocess communications occur. Since the trace evaluation determines all possible traces, the sets S and F will both contain all possible events (in other processes) which may interleave with the events $e^p_{s1} \to \ldots \to e^p_{sn}$ and determining the set difference will eliminate all these events. As expected, the atomic action is local to process P_p.

If any $e^p_{sj} \in C_p$ then interprocess communications do occur and will synchronize both parties to the communication (since $e^p_{sj} \in C_q$ or $e^p_{sj} \in C_r$, as well as C_p). Suppose the communication event concerns processes P_p and P_q. The interprocess communication must be internal to the atomic action. The synchronization it causes will be evident in the trace evaluation. Again since the trace evaluation determines all possible traces, the set S will contain those events in the other process P_q (equivalently P_r) which can immediately precede the first communication with P_p but cannot contain any event which must follow it. Likewise, the set F will contain those events in

process P_q (equivalently P_r) which can immediately follow the last communication with P_p but cannot contain any event which must precede it. Consequently, the set difference operations to give K and J will identify the events in other processes which form the entry line to and the exit line from the atomic action. (Note: the notation J, K, F, S follows from [29])

Algorithm 2 can be used to determine which processes are party to the atomic action.

C. Algorithm 2

1) Determine precise entry and exit lines to the atomic action (sets K and J from algorithm 1).
2) Define the empty set AAP.

$$AAP := \{\}$$

3) For each $g_i \in K$, if $g_i \in \alpha(P_p)$ then add P_p to set AAP.

$$(\forall g_i)\,|\,(g_i \in K) \wedge (g_i = \alpha(P_p)) \rightarrow AAP := AAP \cup P_p$$

(similarly if $g_i \in \alpha(P_q)$ or (P_r))

AAP becomes the set of atomic action processes.

D. Justification of Algorithm 2

Initially before algorithm 1 is executed:

$$AAP = \{\}$$

Building on the justification for algorithm 1, any interprocess communications are recognized by the synchronizing effects they have in the trace evaluation and are constrained to be internal to the atomic action. It is argued above that the sets K and J contain the events which must precede and which must follow the atomic action and identifying the host process for each event is sufficient to determine which processes must be party to the atomic action.

V. Examples

A few simple examples illustrate how the algorithms work. Obviously, the method depends on the correct analysis of interprocess communications, and these examples concentrate on the communications structure.

In the simple program in Fig. 3, P1 communicates with P2 and P2 with P3.

Initially the special events e^p_{init}, e^q_{init}, and e^r_{init} are added before the start of each process. The trace evaluation can then proceed. The trace evaluation proceeds to yield the eventual expansion given in Fig. 4.

Example 1

Suppose it is decided to protect event c1. Then analysis determines:

S = {a1, b1, d1}	since a_{init} always precedes a_1, etc. i.e., the set of all possible immediately preceding "events"
F = {a2, b2, d1}	i.e., the set of all possible immediately following "events"
K = S–F = {a1, b1}	defines those events which should immediately precede the start of the atomic action
J = F–S = {a2, b2}	defines those events which should immediately follow the end of the atomic action
AAP = {P1, P2}	⇒ P3 ∉ AAP

Hence the atomic action enclosing event c1 includes processes P1 and P2, begins immediately after event a1 in process P1 and event b1 in process P2, and terminates immediately before event a2 in process P1 and event b2 in process P2.

Example 2a

Suppose it is decided to protect event c3, say. Then:

S = {a2, b2, d1} K = {b2, d1}

F = {b3, d2, a2} J = {b3, d2} AAP = {P2, P3} ⇒P1 ∉ AAP

Example 2b

To protect c3, b3:

S = {a2, b2, d1} K = {b2, d1}

F = {a2, d2, c4} J = {d2, c4} AAP = {P2, P3} ⇒P1 ∉ AAP

Example 2c

To protect c1, a2, c2:

S = {a1, b1, d1} K = S

F = {a3, d3, b5} J = F AAP = {P1, P2, P3}

Example 3

Suppose it is necessary to identify a boundary in P3 to protect d2, but there is freedom to place it earlier or later in the sequence

```
    𝒫 = (P1 || P2 || P3)
where
    P1 = (a1->c1!->a2->c2?->a3->SKIP)
    P2 = (b1->c1?->b2->c3!->b3->c4?->b4->c2!->b5->SKIP)
    P3 = (d1->c3?->d2->c4!->d3->SKIP)
```

Fig. 3. Simple example with three parallel processes (for simplicity, lower case letters denote CSP events; communications are identified by appending ! or ?).

$$\begin{aligned}
traces(\mathcal{P}) = \{ &\langle t1,c1,t5,c3,t8,c4,b4,c2,t13,\surd^a \| \surd^b \| \surd^d \rangle,\\
&\langle t1,c1,t5,c3,t8,c4,t10,c2,t14,\surd^a \| \surd^b \| \surd^d \rangle,\\
&\langle t1,c1,t7,c3,t9,c4,b4,c2,t13,\surd^a \| \surd^b \| \surd^d \rangle,\\
&\langle t1,c1,t7,c3,t9,c4,t10,c2,t14,\surd^a \| \surd^b \| \surd^d \rangle,\\
&\langle t1,c1,t7,c3,t8,c4,t12,c2,t13,\surd^a \| \surd^b \| \surd^d \rangle,\\
&\langle t1,c1,t7,c3,t8,c4,t11,c2,t14,\surd^a \| \surd^b \| \surd^d \rangle,\\
&\langle t4,c1,t6,c3,t8,c4,b4,c2,t13,\surd^a \| \surd^b \| \surd^d \rangle,\\
&\langle t4,c1,t6,c3,t8,c4,t10,c2,t14,\surd^a \| \surd^b \| \surd^d \rangle,\\
&\langle t4,c1,b2,c3,t9,c4,b4,c2,t13,\surd^a \| \surd^b \| \surd^d \rangle,\\
&\langle t4,c1,b2,c3,t9,c4,t10,c2,t14,\surd^a \| \surd^b \| \surd^d \rangle,\\
&\langle t4,c1,b2,c3,t8,c4,t12,c2,t13,\surd^a \| \surd^b \| \surd^d \rangle,\\
&\langle t4,c1,b2,c3,t8,c4,t11,c2,t14,\surd^a \| \surd^b \| \surd^d \rangle\}
\end{aligned}$$

where, in summary:

t1 = interleavings of a_{init}, b_{init}, a1, b1
<u>or</u> interleavings of a_{init}, b_{init}, d_{init}, a1, b1

t4 = interleavings of a_{init}, b_{init}, d_{init}, a1, b1, d1

t5 = interleavings of d_{init}, a2, b2, d1

t6 = interleavings of a2, b2

t7 = interleavings of d_{init}, b2, d1

t8 = interleavings of b3, d2

t9 = interleavings of a2, b3, d2

t11 = interleavings of a2, b4, d3

t12 = interleavings of a2, b4,

t13 = interleavings of a3, b5, d3

t14 = interleavings of a3, b5

Fig. 4. Final set of traces for example in Fig. 3; recall a_{init} always precedes a_1, b_{init} always precedes b_1, d_{init} always precedes d_1.

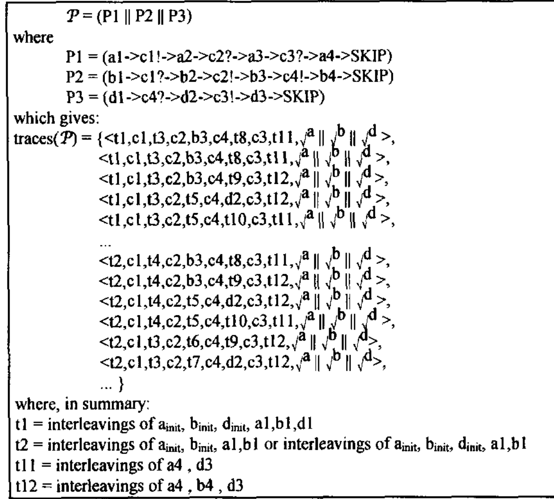


Fig. 5. A further example with three intercommunicating processes; the full trace set comprises 1,488 traces.

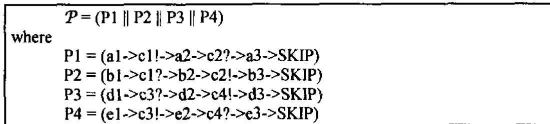


Fig. 6. Simple example with four parallel processes.

c3->d2->c4 in P3; traces($\mathcal{P}$) includes <..., c3, t8, c4, ...> and <..., c3, t9, c4, ...> where t8 = interleavings of b3, d2, and t9 = interleavings of a2, b3, d2.

For d2 alone:

S = {a2, c3, b3} K = {c3}
F = {a2, b3, c4} J = {c4} and AAP = {P3}

However, if the proposed boundary has to enclose c3->d2, then

S = {a2, b2, d1} K = {b2, d1}
F = {a2, b3, c4} J = {b3, c4} and AAP = {P2, P3}

But if the proposed boundary has to include also the communication event c4, thus c3->d2->c4, then:

S = {a2, b2, d1} K = {b2, d1}
F = {a2, b4, d3} J = {b4, d3} and AAP = {P2, P3}

EXAMPLE 4

Consider now the example in Fig. 5. In this system P2 communicates with P3 using c4 and P3 replies to P1 across c3. Suppose it is necessary to protect the sequence c1!->a2->c2?->a3 ->c3? in process P1. Then:

S = {a1, b1, d1} K = S
F = {a4, b4, d3} J = F and AAP = {P1, P2, P3}

EXAMPLE 5

Consider Fig. 6, where $\mathcal{P}$ = (P1 || P2 || P3 || P4) = ((P1 || P2) || (P3 || P4)) and alphabets α (P1 || P2) and α (P3 || P4) have no common event. Clearly, (P1 || P2) is independent of (P3 || P4).

Traces($\mathcal{P}$) must include all possible, arbitrary, interleaving of traces(P1 || P2) and traces(P3 || P4). One possible trace in traces($\mathcal{P}$) must be <any trace from traces(P1 || P2), any trace from traces(P3 || P4)>. Likewise another possible trace must be <any trace from traces(P3 || P4), any trace from traces(P1 || P2)>. This can be determined by explicit trace evaluation.

For any combination of events in (P1 || P2), both S and F must include a trace from traces(P3 || P4), i.e., α (P3 || P4). Hence, the set differences: K = S–F will eliminate α (P3 || P4), and J = F–S will eliminate α (P3 || P4). Hence no events in (P3 || P4) contribute to an atomic action involving events solely in (P1 || P2).

E. Nested Atomic Actions

Atomic actions must be nested correctly and any method for identifying atomic actions must recognize the proper nesting [30]. If a faulty identification is used in the design of software fault tolerant structures, then the scope for error propagation from one atomic action to another may not be eliminated, making error recovery incomplete, or a process may leave an atomic action prematurely making recovery impossible.

Consider two atomic actions AA_a and AA_b, with entry lines defined by K_a and K_b and exit lines defined by J_a and J_b, following the definitions of K and J above. Atomic action AA_a encompasses the sequence of events between K_a and J_a and trace evaluation allows the designer to reason about the relative sequence of events within the different processes within AA_a. Let $\leq$ denote a temporal precedence relationship within a trace.

1) If $(J_a \leq K_b)$ then AA_a happens before AA_b, i.e., $AA_a < AA_b$.
2) If $(J_b \leq K_a)$ then AA_b happens before AA_a, i.e., $AA_b < AA_a$.
3) If $(K_a \leq K_b) \wedge (J_b \leq J_a)$ then AA_b is nested correctly in AA_a, i.e., $AA_a \supseteq AA_b$.
4) If $(K_b \leq K_a) \wedge (J_a \leq J_b)$ then AA_a is nested correctly in AA_b, i.e., $AA_b \supseteq AA_a$.

These are the only conditions that constitute correctly nested atomic actions. Any other set of conditions will produce incorrect nesting.

F. Justification of Nested Atomic Actions

Algorithm 1 produces sets K and J, which identify the events in all the processes which form the entry line and the exit line from the atomic action. In addition, algorithm 2 has been shown to identify which processes are party to the atomic action. Thus, any atomic action, whether nested or not, identified by these algorithms will produce a correct and sufficient set of processes and will identify the entry and exit lines for these actions. It must therefore be the case that these algorithms, defined earlier, identify nested atomic actions correctly.

As an example of this, consider again the system described in Fig. 3. Suppose it is required to identify two atomic actions in this system:

1) to protect c1->c2, in process P1, and
2) to protect c3->c4, in process P3.

This might be attempted (incorrectly) as shown in Fig. 7a. However, by using the algorithms described, the atomic action to protect c3->c4 is given by:

K = {b2, d1} and J = {b4, d3}. (Example 3)

Similarly the atomic action to protect c1->c2 is given by:

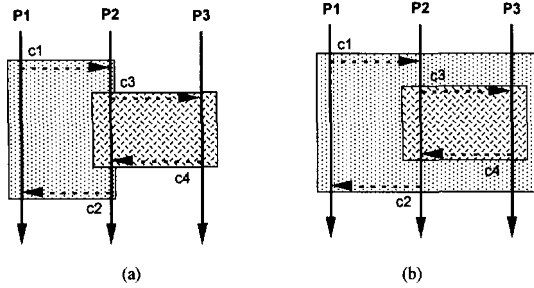


Fig. 7. (a) Incorrect attempt to nest atomic actions; P3 could leave its "atomic action" with P2 before P2 has completed the "atomic action" with P1; (b) Correct nesting: the atomic action between P1 and P2 must include P3 and P3 cannot leave until the atomic action is complete.

$$K = \{a1, b1, d1\} \text{ and } J = \{a3, d3, b5\}. \text{ (Example 2c)}$$

This actually gives the entry and exit lines shown in Fig. 7b. Thus, the outer atomic action encloses the inner action completely, making the nesting correct.

VII. Structural Arguments in Atomic Action Identification

The technique discussed so far requires full trace evaluations to identify atomic action boundaries. However, even with automated tools such as CoPla, the demands on system resources may become too great to allow full trace evaluations. For example CoPla requires 1.7MB of memory for the program, plus approximately (24 × number of traces × average length of trace) bytes for its data structures. Clearly, for real-world systems some means of avoiding full trace evaluation is advisable.

Structural arguments suggest that it is possible in many designs to avoid a full trace evaluation and still recognize atomic action boundaries. By and large the structure of the interprocess communications determines the location of atomic action boundaries. Local events (e.g., logical and arithmetic evaluation, and assignments) are of no interest, nor are constructs governing their sequence (such as loops and conditional selection constructs) if they contain no interprocess communication. Thus, for example, during trace evaluation any sequence of assignments can be collapsed into a single event, simplifying the analysis. It is possible to generate the sets J and K (and thus identify the boundaries) without the need for full trace evaluations.

It is assumed, as earlier, that the designer has chosen a sequence of consecutive events in one process ($e^p_{s1} -> \ldots -> e^p_{sn}$ within process P_p). The sequence must be chosen to enclose fully any internal parallel or selection constructs, and the description must be well-structured (in the sense that it can be translated into an occam implementation). The designer wishes to determine where the atomic action boundary which encloses this sequence must lie. More precisely, the question is which other processes are involved in the atomic action and where does the boundary lie within these processes. The following algorithm determines which events must be included within the atomic action, thereby allowing the designer to define its boundary and to identify all other processes which must be party to the atomic action, without having to produce the complete trace set for the whole system.

A. Algorithm 3

The algorithm marks those events which must be party to the atomic action. Let $\{e^p_{s1}, e^p_{sn}\}$, the set of the first and last events to be protected in process P_p. $\alpha(P_p)$ has its usual meaning as the alphabet of the process P_p. Then, define the primitive functions:

- Type(x) which determines whether its argument, "x," is a local event (localevent), a communication event (comm), a sequential process (SEQ), a parallel process, a choice process or a guarded choice process.
- Value(x) which expects an event as argument and returns its value (i.e., its name).
- Marked(x) which returns a Boolean indicating whether the event or process, "x," has already been recognized as part of the atomic action.
- InsertMark(x) will cause Marked(x) to return TRUE on its next call.
- List(N) which is an ordered list of the events in the process given as its argument (effectively the trace restricted to the events in process N).

The following auxiliary functions simplify the analysis:

- Mark(a) modifies the marked attribute of its argument, "a," setting it to the value TRUE; if the element is a communication event then the other participant is also marked.

  Mark(a): $\overline{(\text{Marked}(a))} \rightarrow (\forall x) \big| (x \in (\alpha(a)))$
  1) InsertMark(x);
  2) (Type(x) = comm)-> (y := Partner(x));
  Mark(y);
  3) result := TRUE;

- Partner(a), given a communication event "a" as argument, returns the other participant.

  Partner(a) : $(\exists x) \mid (x \in (\alpha(P)))$
  $(\text{Type}(x) = \text{comm}) \wedge (\text{Value}(x) = \text{Value}(a)) \wedge (x \neq a)$->result := x;

- SequentiallyPostDependent(a, b, N) determines whether event "b" is sequentially post dependent on "a"; in other words whether "b" must necessarily occur after "a" has finished. (x > y means that "x" occurs after "y" in the ordered list of N).

- SequentiallyPostDependent (a, b, N):
  $(a \in \alpha(N)) \wedge (b \in \alpha(N)) \wedge$
  $(\text{Type}(N) \neq \text{localevent}) \wedge (\text{Type}(N) \neq \text{comm})$->
  $(\forall x) \mid (x \in (\text{List}(N)))$
  $(a \in (\alpha(x))) \wedge (b \in (\alpha(x)))$->
  result := SequentiallyPostDependent (a, b, x)
  $(a \in (\alpha(x))) \wedge (b \notin (\alpha(x))) \wedge (\text{Type}(N) = \text{SEQ})$->

$(\exists y) \mid (y \in (List(N))) \wedge ((b \in \alpha(y)) \wedge (x > y))$->
result := TRUE;

- Then the function <u>FindAA</u>:
  - Step 1) Marks all the events in the set of events E which have to be protected.
  - Step 2) Ensures that all events between any two events that have to be executed in sequence are included within the boundary. For each marked event x in P, for each marked event y which is sequentially post dependent on x, mark all the events which are sequentially between x and y.
  - Step 3) Ensures that all processes can exit at the same time.
    For each marked event x in P:
    - (a) define the empty set H. Insert into H all nonmarked communication events y that are sequentially post dependent of x. Include both participants.

    For each element y of the set H, for each event z that is sequentially post dependent of y:
    - (b) if z is marked then mark y. If there has been any newly marked event then reiteration of step 2 is needed.
    - (c) if z is not marked and it is a communication event then if z or its partner do not belong to H include them and reiterate step 3b.
- FindAA():

Step 1) $(\forall x) \mid (x \in E) \rightarrow Mark(x)$
Step 2)
a) $(\forall x) \mid (x \in (\alpha(P)))$
$(\forall y) \mid (y \in (\alpha(P)))$
$(Marked(x)) \wedge (Marked(y)) \wedge$
$(SequentiallyPostDependent\ (y, x, P)) \rightarrow$
$(\forall z) \mid (z \in (\alpha(P)))$
$(SequentiallyPostDependent(z, x, P)) \wedge$
$(SequentiallyPostDependent(y, z, P)) \rightarrow$
$(Mark(z)) \rightarrow goto2 := TRUE;$
b) $(goto2) \rightarrow$ go to step 2a
Step 3) $(\forall x) \mid (x \in (\alpha(P))) \wedge (Marked(x))$
a) $H := \{\};$
$(\forall y) \mid (y \in (\alpha(P)))$
$\overline{(Marked(y))} \wedge (Type(y) = comm) \wedge$
$(SequentiallyPostDependent(y, x, P)) \rightarrow$
$H := H \cup y \cup Partner(y);$
b) $(\forall y) \mid (y \in (H))\ (\forall z) \mid (z \in (\alpha(P)))$
$(SequentiallyPostDependent(z, y, P)) \wedge$
$(Marked(z)) \rightarrow Mark(y) \rightarrow goto2 := TRUE;$
c) $(\forall y) \mid (y \in (H))\ (\forall z) \mid (z \in (\alpha(P)))$
$(SequentiallyPostDependent(z, y, P)) \wedge$
$\overline{(Marked(z))} \wedge (Type(z) = comm) \wedge ((z \notin H) \vee$
$(Partner(z) \notin H)) \rightarrow H := H \cup z \cup Partner(z);$
$goto3.b := TRUE;$
d) $(goto3.b) \rightarrow$ go to step 3.b
Step 4) $(goto2) \rightarrow$ go to step 2

The algorithm seeks out communication patterns amongst the set of processes in an iterative fashion and deliberately examines the sequential dependence of communications in the system. It uses this sequential dependence to examine not only all direct communications with the original sequence requiring protection, between the events in set E, but also any subsequent communications (set H) from processes with which events in E have had contact. It determines whether these subsequent communications have structural implications which require their inclusion in the atomic action. The algorithm'iterates until no further events are identified for inclusion in the atomic action.

The algorithm progressively marks those events in the complete set of processes which should be included in the atomic action. The final step is to generate the set AAe which identifies the events constituting the atomic action, and the set AAP to determine which processes are necessarily party to the atomic action.

1. Define the empty set AAe
   $AAe := \{\}$
2. Add each marked event to AAe
   $(\forall x) \mid (x \in (\alpha(P))) \wedge (Marked(x))$->
   $AAe := AAe \cup x$
3. Define the empty set AAP
   $AAP := \{\}$
4. For each event x in AAe, for all processes P_p,
   if $e \in \alpha(P_p)$ then add P_p to set AAP.
   $(\forall e)\ (\forall P_p) \mid (e \in AAe) \wedge (e = \alpha(P_p))$->
   $AAP := AAP \cup P_p$

These structural algorithms have been applied to a large number of examples.

VIII. Examples

Consider again the set of processes given in Fig. 3:

$\mathcal{P}$ = (P1 || P2 || P3)

where

P1 = (a1->c1!->a2->c2?->a3->SKIP)
P2 = (b1->c1?->b2->c3!->b3->c4?->b4->c2!->b5->SKIP)
P3 = (d1->c3?->d2->c4!->d3->SKIP)

Atomic actions can be readily identified <u>without</u> the need to evaluate the traces given in Fig. 4.

Example 1 Revisited

Suppose it is decided to protect event c1. Hence E = {c1}. Then step 1 causes the following marking of events ($\square$ indicates marked events).

P1 = (a1->$\boxed{c1!}$->a2->c2?->a3->SKIP)
P2 = (b1->$\boxed{c1?}$->b2->c3!->b3->c4?->b4->c2!->b5->SKIP)
P3 = (d1->c3?->d2->c4!->d3->SKIP)

Step 2 leads to no new markings, and consequently the algorithm gives AAe = {c1} and AAP = {P1, P2}. In other words the atomic action enclosing event c1 includes only the event c1 in both processes P1 and P2. This is consistent with the earlier analysis that the atomic action begins immediately after event a1 in process P1 and event b1 in process P2, and terminates immediately before event a2 in process P1 and event b2 in process P2.

EXAMPLES 2A, 2B, 2C REVISITED

In a similar way, these produce analogous results to those produced earlier using algorithm 1.

EXAMPLE 3 REVISITED

Here the designer has the opportunity of selecting to protect either d2 alone, but to include other events in the sequence c3 -> d2->c4 in P3. For E = {d2}, algorithm 3 quickly terminates with AAe = {d2} as the sole constituent of the atomic action. However, if E = {c3, d2} were selected, then step 1 would cause the following marking:

P1 = (a1->c1!->a2->c2?->a3->SKIP)

P2 = (b1->c1?->b2->c3! ->b3->c4?->b4->c2!->b5->SKIP)

P3 = (d1->c3?->d2 ->c4!->d3->SKIP)

giving AAe = {c3, d2} and AAP = {P2, P3}.

Likewise, if E = {d2, c4} were selected, then step 1 would cause the following marking:

P1 = (a1->c1!->a2->c2?->a3->SKIP)

P2 = (b1->c1?->b2->c3!->b3->c4? ->b4->c2!->b5->SKIP)

P3 = (d1->c3?->d2->c4! ->d3->SKIP)

giving AAe = {d2, c4} and AAP = {P2, P3}.

However, if E = {c3, c4} were selected, then step 1 and step 2 would cause the following marking:

P1 = (a1->c1!->a2->c2?->a3->SKIP)

P2 = (b1->c1?->b2->c3!->b3->c4? ->b4->c2!->b5->SKIP)

P3 = (d1->c3?->d2->c4! ->d3->SKIP)

giving AAe = {c3, d2, c4, b3} and AAP = {P2, P3}.

EXAMPLE 4 REVISITED

This example concerns the set of processes:

$\mathcal{P}$ = (P1 || P2 || P3)

where

P1 = (a1->c1!->a2->c2?->a3->c3?->a4->SKIP)

P2 = (b1->c1?->b2->c2!->b3->c4!->b4-->SKIP)

P3 = (d1->c4?->d2->c3!->d3->SKIP)

As before, suppose it is decided to protect the sequence c1!->a2->c2?->a3->c3? in process P1, i.e., E = {c1, c3}. Then step 1 causes the marking:

P1 = (a1->c1! ->a2->c2?->a3->c3? ->a4->SKIP)

P2 = (b1->c1? ->b2->c2!->b3->c4!->b4-->SKIP)

P3 = (d1->c4?->d2->c3! ->d3->SKIP)

Step 2 now causes the marking:

P1 = (a1->c1!->a2->c2?->a3->c3? ->a4->SKIP)

P2 = (b1->c1? ->b2->c2! ->b3->c4!->b4-->SKIP)

P3 = (d1->c4?->d2->c3! ->d3->SKIP)

This step is reiterated, since new events were marked:

P1 = (a1->c1!->a2->c2?->a3->c3? ->a4->SKIP)

P2 = (b1->c1?->b2->c2! ->b3->c4!->b4-->SKIP)

P3 = (d1->c4?->d2->c3! ->d3->SKIP)

Now step 3a looks at the unmarked communications events, to form the set H = {c4!, c4?}. Step 3b would discover that c3! in process P3 is marked and sequentially post dependent, leading to the marking:

P1 = (a1->c1!->a2->c2?->a3->c3? ->a4->SKIP)

P2 = (b1->c1?->b2->c2! ->b3->c4! ->b4-->SKIP)

P3 = (d1->c4? ->d2->c3! ->d3->SKIP)

Reiteration of step 2 then leads to the marking:

P1 = (a1->c1!->a2->c2?->a3->c3? ->a4->SKIP)

P2 = (b1->c1?->b2->c2!->b3->c4! ->b4-->SKIP)

P3 = (d1->c4?->d2->c3! ->d3->SKIP)

No further markings are generated by the remaining steps, leading to the conclusion that AAe = {a2, a3, c1, c2, c3, c4, b2, b3, d2} and AAP = {p1, p2, p3}.

B. Complexity of the Algorithms

The full trace algorithm (algorithm 1) would show exponential complexity with the number of processes during trace production if there were no synchronizing communications present. When communications are added, each communication forces synchronization between two processes, eliminates part of the trace set, and thus reduces complexity. Every communication reduces the size of the trace set significantly, and similarly the time required to search. Thus:

- Given n processes, each with m events, and 0 comms: require about n^m traces.
- Given n processes, each with m events, and 1 comms: require about $2.n^{m/2}$ traces.
- Given n processes, each with m events, and p comms: require about $2^p.n^{m/p}$ traces.

Searching is approximately linear with the size of the trace set, since the algorithm is simply scanning the trace sets.

In algorithm 3, this initial complexity does not appear as traces are not explicitly produced. Instead, complexity arises in searching across communication links to identify atomic action boundaries. Algorithm 3 shows a near linear complexity, but whenever sequential post dependency forces backtracking the analysis becomes less obvious. If there were no backtracking, then the complexity would be linearly dependent on the number of events (n*m). Every time the algorithm has to backtrack, it is effectively analogous to regenerating a further set of traces to search. Thus, if there are q backtracking occurrences, then the complexity increases to n*m*q.

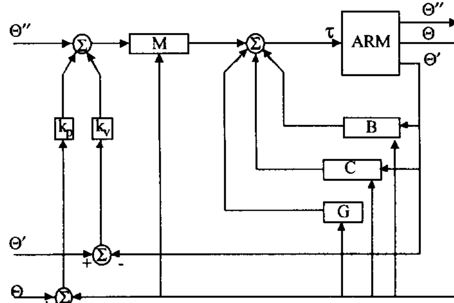
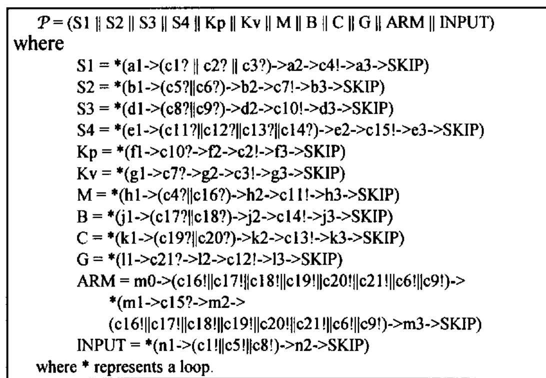


Fig. 8. Model of a robot arm manipulator.



```
P = (S1 || S2 || S3 || S4 || Kp || Kv || M || B || C || G || ARM || INPUT)
where
        S1 = *(a1->(c1? || c2? || c3?)->a2->c4!->a3->SKIP)
        S2 = *(b1->(c5?||c6?)->b2->c7!->b3->SKIP)
        S3 = *(d1->(c8?||c9?)->d2->c10!->d3->SKIP)
        S4 = *(e1->(c11?||c12?||c13?||c14?)->e2->c15!->e3->SKIP)
        Kp = *(f1->c10?->f2->c2!->f3->SKIP)
        Kv = *(g1->c7?->g2->c3!->g3->SKIP)
        M = *(h1->(c4?||c16?)->h2->c11!->h3->SKIP)
        B = *(j1->(c17?||c18?)->j2->c14!->j3->SKIP)
        C = *(k1->(c19?||c20?)->k2->c13!->k3->SKIP)
        G = *(l1->c21?->l2->c12!->l3->SKIP)
        ARM = m0->(c16!||c17!||c18!||c19!||c20!||c21!||c6!||c9!)->
              *(m1->c15?->m2->
              (c16!||c17!||c18!||c19!||c20!||c21!||c6!||c9!)->m3->SKIP)
        INPUT = *(n1->(c1!||c5!||c8!)->n2->SKIP)
  where * represents a loop.
```

Fig. 9. An outline description of the robot arm manipulator.

An explicit indication of the memory requirements for the full trace algorithm was given earlier (1.7MB of memory for the CoPla program, plus approximately (24 × number of traces × average length of trace) bytes for data structures; in the real world robot example discussed below, there are around 180,000 traces). Algorithm 3 only uses a small fraction of this as the algorithm only stores the sets E, J, K, etc. as they are built up, no large sets of traces are generated (for the real world robot example, sets with no more than 120 members have proved adequate).

IX. A Real World Example

Simple examples as shown above can have their trace sequences evaluated "by hand" and boundary identification can be achieved by inspection of the trace sets. For larger, more realistic examples manual methods become unmanageably complex. The CoPla software tool has been used to produce trace sets for a number of more complex systems and then used to identify atomic action boundaries. One application, a robot arm manipulator, (Fig. 8) has been modeled [25] and implemented as a set of 12 parallel processes. A slightly simplified version expressed in CSP is shown in Fig. 9; following initialization, each process engages in an infinite loop, all synchronized by communications. (The CSP description expands to an actual implementation comprising about 10,000 lines of occam code). Automated analysis reveals 184,900 possible traces; CoPla then allows the user to propose entry and exit points for an atomic action within one of the processes and uses the algorithms described earlier to determine the proper boundaries of the atomic action.

For example, consider the requirement to locate an atomic action boundary which encloses c15 to c9 inclusively in process ARM. CoPla gives the following results:

$E_{s2} = \{c6\}$ $E_{s3} = \{c9\}$ $E_{s4} = \{c15\}$ $E_M = \{c16\}$
$E_B = \{c17, c18\}$ $E_C = \{c19, c20\}$ $E_G = \{c21\}$
$E_{ARM} = \{c15, m2, c16, c17, c18, c19, c20, c21, c6, c9\}$

allowing the sets K and J to be derived as:

- K = {b1, d1, e2, h1, j1, k1, l1};
  events which must precede the atomic action boundary, and
- J = {b2, d2, e3, h2, j2, k2, l2};
  events which must follow the atomic action boundary.

X. Conclusions

This paper has proposed two methods for identifying atomic actions in systems described using CSP. If explicit trace evaluation is tractable, then algorithms 1 and 2 provide the designer with a systematic method of locating atomic action boundaries in a hierarchical fashion, essentially by analyzing the possible sequences of interprocess communications within the trace sets. The second method (algorithm 3) takes the original CSP descriptions of the system and uses structural arguments to identify the atomic action boundaries; this method does not suffer the drawbacks involved in full trace evaluation, but does incur the penalty of a more complex algorithm.

Both techniques identify those events which are constituent to a proposed atomic action and eliminate all processes that are disjoint from the atomic action; both techniques allow nested atomic actions to be identified correctly. However, an analysis based on structural arguments has a number of attractions. By avoiding a full trace evaluation or a full reachability analysis, the method is more economical on computational time and memory resources. But, it depends implicitly on the ability to analyze the sequence in which events could occur, which is akin to the ability to generate the complete set of traces. It cannot therefore be used with an arbitrary set of communicating processes; the designer is restricted to processes formed solely from sequential, parallel, conditional and general choice constructs of simple events and communications. Nevertheless, the algorithms have been applied to systems which include restricted forms of program loops. For example, the robot manipulator arm processes are normally invoked from within an infinite control loop, but since the iterations begin and end synchronously, the analysis can be applied without prejudice.

In the examples shown in the paper the processes only have one trace. However, multiple traces (and thus some form of nondeterminism) are easily included into the design by considering each alternative individually, although this will obviously increase the overall number of traces that will

be produced. When the problem of atomic action placement is addressed, for such situations, the main point is that the complete "nondeterministic" structure (such as an occam ALT) must be included in the atomic action.

Both methods operate directly on the CSP description of the system. They require no error prone translation of a developed program into graphical form, nor is there an implied simulation of program execution based on the graphical structures. Furthermore, translation of the CSP design to an occam implementation is generally straightforward (since problematic features such as interrupts are excluded) because of the close family relationship between occam and CSP, or alternatively, hardware implementations can be developed directly from the CSP design with only modest difficulty.

The underlying motivation of this research is to develop a mechanism for introducing software fault tolerance structures in a systematic, proper, fashion. Atomic action identification is just the first, crucial, step in that process.

ACKNOWLEDGMENTS

The authors would like to thank Oscar Saiz for his work on implementing some of the algorithms presented in this paper. Thanks also to the anonymous reviewers for their constructive comments which have helped improve the paper.

REFERENCES

[1] MOD(UK) Interim Defence Standards 00-55 and 00-56, no. 1, Apr. 1991.

[2] "Software considerations in airborne systems and equipment certification," RTCA/D, 178A, RTCA, Washington, DC, 1985.

[3] "Software for computers in the application of industrial safety-related systems," IEC draft standard 65A (Secretariat) 94, Document 89/33006, BSI, 1989.

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

[5] P.A. Lee, and T. Anderson, *Fault Tolerance: Principles and Practice*. Springer Verlag, 1991.

[6] B.H Liskov and R. Scheifler, "Guardians and actions: Linguistic support for robust, distributed programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 3, pp. 381–404, July 1983.

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

[8] T. Anderson and J.C. Knight, "A framework for software fault tolerance in real-time systems," *IEEE Trans. Software Engineering*, vol. 9, no. 12, pp. 355–364, May 1983.

[9] C.A.R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[10] L.V. Mancini, and S.K. Shrivastava, "Replication within atomic actions and conversations: A case study in fault-tolerance duality," *FTCS-19*, Chicago, pp. 454–461, June 1988.

[11] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Engineering*, vol. 1, pp. 220–232, June 1975.

[12] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Trans. Software Engineering*, vol. 11, no. 12, pp. 1,491–1,501, Dec. 1985.

[13] R.K. Scott, J.W. Gault, and D.F. McAllister, "Fault-tolerant software reliability modeling," *IEEE Trans. Software Engineering*, vol. 13, no. 5, pp. 583–592, May 1987.

[14] K.H. Kim, and H.O. Welch, "Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications," *IEEE Trans. Computers*, vol. 38, no. 5, pp. 626–636, May 1989.

[15] E. Best, and B. Randell, "A formal model of atomicity in asynchronous systems," *Acta Informatica*, vol 16, pp. 93–124, 1981.

[16] K.H. Kim, S.M. Yang, and M.H. Kim, "Implementation of concurrent programming language facilities supporting conversation structuring," *Proc. IEEE COMPSAC '85*, pp. 445–453, 1985.

[17] K.H. Kim, "Programmer-transparent coordination of recovering concurrent processes: philosophy and rules for efficient implementation," *IEEE Trans. Software Engineering*, vol 14, no. 6, pp. 810–821, June 1988.

[18] K.H. Kim and S.M. Yang, "Performance impact of look-ahead execution the conversation scheme," *IEEE Trans. Computers*, vol. 38, no. 8, pp. 118–1,202, Aug. 1989.

[19] R.H. Campbell, T. Anderson, and B. Randell, "Practical fault tolerant software for asynchronous systems," *Proc. SAFECOM '83*, Cambridge, pp. 59–65, 1983.

[20] G.F. Carpenter, "The use of Occam and Petri nets in the simulation of logic structures for the control of loosely coupled distributed systems," *Proc. UKSC Conference on Computer Simulation (UKSC-87)*, Bangor, Sept. 1987. Pub. Soc. Computer Simulation, pp. 30–31, Sept. 1987.

[21] G.F. Carpenter and A.M. Tyrrell, "The use of GMB in the design of robust software for distributed systems," *Software Engineering J.*, vol. 4, pp. 268–282, Sept. 1989.

[22] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[23] Inmos, *Occam 2 Reference Manual*. Prentice Hall, 1988.

[24] A.M. Tyrrell, and A.C.A. Smith, "A parallel module for fault tolerant industrial control applications," *IFAC Symp. Parallel and Distributed Computing*, Greece, pp. 205–210, June 1991.

[25] A.M. Tyrrell, and I.P.W. Sillitoe, "Evaluation of fault tolerant software structures for parallel systems in industrial control," *IEE Int. Conf. CONTROL '91*, Edinburgh, pp. 393–398, Mar. 1991.

[26] G.M. Reed, and A.W. Roscoe, "A timed model for CSP," *Theoretical Computer Science*, vol. 58, pp. 249–261, 1987.

[27] Z. Chaochen, "The consistency of the calculus of total correctness for communicating processes," Oxford Univ. Research Group Monograph PRG 26, Feb. 1982.

[28] O.J. Saiz, and A.M. Tyrrell, "Analysis tool for parallel systems," *Proc. First Euromicro Int'l Workshop on Parallel and Distributed Processing*, Gran Canaria, Jan. 27–29, 1993, IEEE Computer Society Press, pp. 499–505, Jan., 1993

[29] A.M. Tyrrell, and D.J. Holding, "Design of reliable software in distributed systems using the conversation scheme," *IEEE Trans. Software Engineering*, vol. 12, no. 7, pp. 921–928, Sept. 1986.

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

Andrew M. Tyrrell received a first class honors degree in 1982 and a PhD in 1985, both in electrical and electronic engineering. He joined the Electronics Department at York University in April 1990. Previous to that, he was a senior lecturer at Coventry Polytechnic. Between August 1987 and August 1988 he was visiting research fellow at Ecole Polytechnic, Lausanne, Switzerland, where he was researching into the evaluation and performance of multiprocessor systems. From September 1973 to September 1979 he worked for STC at Paignton Devon on the design and development of high frequency devices. He is currently head of the Parallel and Signal Processing Research Group at York.

His main research interests are in the design of parallel systems, fault tolerant design, software for distributed systems, parallel systems for numerical problems, real-time simulation using parallel computers and real-time systems. In the last five years he has published over 40 papers in these areas, and has attracted funds in excess of £250,000.

Dr. Tyrrell is a member of the IEE, the IEEE and the ACM.

Geof F. Carpenter is a lecturer in the Department of Electronic Engineering and Applied Physics at Aston University. His long-standing research interests in the design of real-time, distributed computing systems have been directed towards applications where high reliability and safety are essential requirements, such as in high-speed flexible manufacturing systems. Since appointment to Aston University in 1980, Dr. Carpenter has been involved in a large number of courses in information systems engineering, and he now has wide responsibility for handling undergraduate admissions to, and student progression through, degree programs in electronics and computing. Dr. Carpenter is a chartered engineer, and a member of the Institution of Electrical Engineers and the British Computer Society.