A UNIFIED MODEL FOR CONCURRENT DEBUGGING †

S. I. Hyder, J. F. Werth and J. C. Browne‡
The University of Texas at Austin
Austin, Texas 78712

Abstract: *Events are occurrence instances of actions. The thesis of this paper is that the use of "actions", instead of events, greatly simplifies the problem of concurrent debugging. Occurrence instances of actions provide a debugger with a unique identifier for each event. These identifiers help the debugger in recording the event orderings. The recorded orderings indicate much more than a mere temporal order. They indicate the dependences that "cause" the actions to execute. A debugger can, then, collect the dependence information from the orderings of different instances of the same action, and deduce the conditions that govern the execution of the action. This provides a framework for representing and checking the expected behavior. Unlike existing approaches, we cover all parts of the debugging cycle. Our unified model, therefore, allows a single debugger to support different debugging facilities like execution replay, race detection, assertion/model checking, execution history displays, and animation.*

Keywords: Pomset, concurrent debugging, execution replay, race detection, animation, model checkers.

1.0 INTRODUCTION

Parallel programs typically express concurrency by adding synchronization constructs to the usual sequential text. This produces a complex entanglement of the concurrent considerations with the sequential considerations. A programmer, then, has to debug the synchronizations and communications in the presence of the already complex problem of debugging the flow of data and the flow of control of the sequential text. This makes the debugging of concurrent programs an extremely complex problem.

During debugging, a programmer forms some expectations about the execution behavior of the program. This expected behavior is, then, represented by an assertion/model. Checking of the actual execution against this assertion/model, reveals any unexpected behavior. Mapping of this behavior to the program, brings the programmer closer to the bug. A sequence of these interactions between the specified behavior of the program P, model of the expected behavior M, and the observed execution behavior E constitutes a debugging cycle. This cycle will be illustrated as P→ M→E → P.

Concurrent debugging becomes so complex because of the ambiguities that obscure the interactions between P, M and E in the debugging cycle. These ambiguities arise from the entanglement of the concurrent and sequential considerations in the representations used for P, M and E. As these ambiguities obscure the interactions, each part of the debugging cycle becomes a separate problem area. Existing debugging facilities that target separate parts of the debugging cycle, then, appear incompatible and even orthogonal. Incompatibility of these facilities forces the programmer to either use different facilities for different parts of the cycle or debug without them. On the other hand, it compels the debuggers to either constrain the range of behaviors that can be checked [9], [7]; or to tolerate the ambiguities in the observed behavior [8], [16]; or to demand extra programming effort [4], [12], [6], [1].

Def. 1: A *computation action* is a piece of program text that starts and/or ends with a synchronization statement.

Although concurrent debuggers define execution events to be the occurrence instances of program actions, they often leave the actions implicit. The thesis of this paper is that the complexity of concurrent debugging greatly simplifies when the set of actions, instead of events, is used to define the various behaviors used in debugging. Our unified model of concurrent debugging debugs the concurrent behavior using the relations on the set of computation actions; specified in the program, modeled in the expected behavior and observed in the execution behavior.

We use the abstraction of computation actions and the "causality" of their dependence relations to disentangle the concurrent considerations from the sequential. This decomposes the concurrent debugging problem into two almost disjoint problems that can be debugged at different levels. A programmer debugs the concurrent state (§ 3.3) at the upper level, where the only important concerns are the relations on the set of computation actions. Internal states of a computation action are not important at this level. They only become important when the programmer moves to the lower level, inside the action.

The set of computation actions serves as a basis for instrumenting the program. The debugger identifies the events as occurrence instances of actions. As an action can occur "multiple" number of times, the identity of the action and its instance number act together as a

† This work is supported in part by the research initiation grant program of IBM, Corp., to support interdesciplinary computer science research.

‡ Department of Computer Sciences.
E-mail: {hyder | jwerth | browne}@cs.utexas.edu.

unique identifier (logical clock) for each event. Using these identifiers, our debugger records a partial order that we call the causal orderings (§ 3.1). In this ordering, the immediate predecessors and successors of a given event map to the dependences of the action corresponding to the event (§ 3.2).

A programmer's expectations consist of a set of events and an orderings on these events. An execution is erroneous if the expected events do not occur, or occur in some un-expected order. The programmer, therefore, describes the expected behavior with actions and the dependences that order the occurrences of those actions as events (§ 4.1). If the observed ordering relations do not match the conditions represented in the expected behavior, then a debugger has detected some unexpected behavior and raises the exception (§ 4.2).

Our approach differs from the existing ones in that it uses actions, instead of events. It is program oriented, instead of execution oriented. It records the causal orderings, instead of approximating them. It unifies all the parts of the debugging cycle. This overcomes the incompatibility of existing facilities, and allows one debugger to support different debugging facilities like execution replay, race detection, assertion/model checking, execution history displays, and animation.

1.1 Problems in Various Parts of the Cycle

Complexity of concurrent debugging has received much attention in recent years. A 1989 bibliography cited over 370 references [21]. Existing approaches complicate the problem by only covering a subpart of the debugging cycle; P → M → E → P. This makes them incompatible, and compels the user to debug the remaining parts of the cycle with separate debugging facilities. This often involves extra programming effort.

Execution environments typically provide processes or threads as executable units to run the program text. There is often a complex multiplexing of the program text among process/thread structure due to resource limitations, scheduler policies, and other constraints. Debugging support must resolve the timing ambiguities arising from such multiplexings. An *execution history display* [12], [4], [20], [14] helps in resolving some of these ambiguities. It, however, only provides a time-process graph representation of E. An *animation facility* [20], [6] is, then, needed to provide an instantaneous view of the E → P mapping. A textual representation of P is inadequate for supporting this mapping. Hence, extra user effort is needed to develop a graphical structure that can support animation.

Moreover, another debugger is needed to resolve the ambiguities arising from the inability to record the causal orderings in P → E part of the cycle [8], [18].

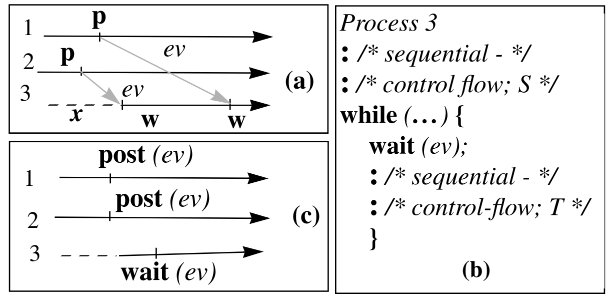


FIGURE 1. (a) A time-process graph (b) Program text segment for process 3 (c) Event traces.

Such a debugger only helps in modeling the race behaviors [18], and in automating *race detection*. An *assertion/model checker* [1] is, then, needed to help in modeling the expected behavior, and in automating its checking. However, a textual representation of P makes it difficult for the user to represent the expected behavior in P → M part of the cycle. The text does not allow the user to represent conditions about the concurrent state that involve event orderings. Hence, execution and *problem* oriented approaches [7] are used. They, however, demand extra user effort [1]. Furthermore, their use of events, instead of actions, creates additional problems in recognizing the expected behavior in M → E part of the cycle.

Finally, an execution replay facility is needed to make *cyclical* debugging possible [10].

Mapping Ambiguities: In E → P part of the cycle, a debugger has to map the events defined in the context of processes/threads of the execution environment on to the program text. However, ambiguities arise in mapping intra-process arcs of a time-process graph to their corresponding sequential text in the program. For instance, in Fig. 1(a) there is an ambiguity about the intra-process arc ***x*** of Process 3. ***x*** can either map to the sequential text S or to the sequential text T of Fig. 1(b). Event ***w*** that immediately follows ***x***, and maps to the synchronization statement **wait** *(ev)* is not of much help. **wait** *(ev)* is neither associated with S nor with T. The synchronization event simply sits at the boundary where a piece of text ends and another one starts.

Instead of letting a synchronization statement sit ambiguously on the border of two sequential text segments, we propose an abstraction that permanently associates the synchronization statements of a program with its sequential text segments. The abstractions resulting from this association is that of a *computation action* (Appendix). It disentangles the sequential control-flow considerations from the synchronization considerations.

Animation facilities [14], [20] often demand extra user effort to develop an alternate structure for supporting their visualizations of the E $\rightarrow$ P mapping. A textual representation of P can not support this visualization because it conceals the synchronization dependences on which inter-process arcs of a time-process graph map to. These dependences are concealed in the semantics of the synchronization constructs. For e.g., the dependences that order **p** and **w** in Fig. 1(a), are concealed in the semantics of **wait** *(ev)* in Fig. 1(b) that shows the text of process 3.

We, therefore, use a graphical representation of P [15] whose nodes are the computation actions and whose arcs are their dependences (§ 2.0). Occurrence instances of computation actions are partially ordered. They provide a "pomset" representation of E that allows us to automatically generate the animation structure (§ 3.0).

Ordering Ambiguities: In P $\rightarrow$ E part of the cycle, a debugger should record the events and their orderings. Distributed systems often record the event orderings by exploiting the data dependences introduced by the send/receive of messages with the help of unique time-stamps (or identifiers) [13]. Shared memory debuggers that detect races [16], [8], however, ignore the data dependences introduced by the accesses to the shared synchronization variables. They, also, ignore the importance of unique identifiers or time-stamps for each event. Their recorded event traces, therefore, contain ambiguities about the inter-process orderings as shown in Fig. 1(c). There is ambiguity as to whether the **wait** *(ev)* of process 3 was fulfilled by the **post** *(ev)* of process 1 or 2. This necessitates the use of approximations [8], and leads to intractability [17]. Inability to record the order of accesses to shared objects further complicates the detection of *races* (simultaneous access to shared objects with at least one write), and affects the accuracy of detected races [16], [18].

Note, however, that if an event that write-accesses a shared object, appends its unique identifier to the object, then a later event access to that object can identify its "causal" predecessor (§ 3.1). This observation allows us to support execution replay, and to support race detection without any extra overhead (§ 5.0).

Modeling Problems: A *dbx* debugger is user-friendly. It allows the user to associate expected conditions with a program action. For instance, in a *dbx* command like "**when at** *stmt* **if** *condition*", the user associates a condition about the sequential state with a statement of interest. Later, the debugger allows the user to interactively follow the conditional progress of the execution that has been restricted to the interesting actions. Such a *program oriented* approach [7] is not possible with a "textual" representation of the concurrent program. Unlike *dbx* conditions, conditions about the concurrent state involve event orderings whose corresponding dependences are not visible in the textual representation. For instance, event orderings in an expected behavior like "($\boldsymbol{w} \wedge \boldsymbol{p}$) precede $\boldsymbol{w}$" for Fig. 1(a), correspond to the dependences that are not visible in the textual representation of Fig. 1(b). Hence, assertion/model checkers [7], [14] adopt execution oriented approaches that use models like temporal logic, interleaving, partial order or automatas [9]. In P $\rightarrow$ M part of the cycle, therefore, a user has to exert extra effort to learn a new language, program the expected behavior and, then, debug it for *user errors* [1].

Filtering Ambiguities: In M $\rightarrow$ E part of the cycle, *assignment* and *resolution* problems [1] are typical of the ambiguities that arise during filtering and recognition of the expected behavior. As existing checkers are execution oriented, they use events in their representation of the expected behavior, and leave the actions implicit. This conceals the information that (i) events are actually multiple occurrences of actions, and (ii) the observed event orderings are the unrolling of the communication/synchronization structure of the program actions. For instance, a behavior like "**p** precedes **w**" gives no information to the debugger about the actions that correspond to events **p** and **w**. Ambiguities can, then, arise whenever more than one observed behavior fits the expected behavior. In Fig. 1(a), "**p** precedes **w**" can fit several behaviors; **p** of process 2 precedes the first **w** of process 1, **p** of process 1 precedes the second **w** of process 2, or **p** of process 2 precedes the second **w** of process 3. Such ambiguities restrict the range of checkable behaviors. This, in turn, restricts the range of behaviors that can be represented in M [9], [1].

The information about the actions like their statement line number or process id, could have resolved these ambiguities. We, therefore, use such information about the actions to simplify the filtering and recognition of the expected behavior (§ 4.2).

1.2 Example: Events as Occurrences of Actions

Fig. 2(b) graphically shows the actions $\boldsymbol{w}$, $\boldsymbol{p}$, and $\boldsymbol{q}$, and their dependences that we denote by the ordered pairs $(\boldsymbol{w}, \boldsymbol{w})$, $(\boldsymbol{p}, \boldsymbol{w})$, and $(\boldsymbol{q}, \boldsymbol{w})$. This is formalized later. $\boldsymbol{i}$-th occurrence instances of the actions give us event identifiers $\boldsymbol{w_i}$, $\boldsymbol{p_i}$, and $\boldsymbol{q_i}$. Our debugger records the event orderings by appending the identifiers to the shared state of the dependences (§ 3.1). The observed orderings, therefore, indicate those dependences of actions that "caused" the events to be ordered. So, event orderings of Fig. 2(a) indicate that they were caused by the dependences of Fig. 2(b): Ordering $\boldsymbol{p_1} < \boldsymbol{w_3}$ was caused by the dependence $(\boldsymbol{p}, \boldsymbol{w})$, ordering $\boldsymbol{q_1} < \boldsymbol{w_2}$ was caused by the dependence $(\boldsymbol{q}, \boldsymbol{w})$, and $\boldsymbol{w_1} < \boldsymbol{w_2}$ and $\boldsymbol{w_2} < \boldsymbol{w_3}$ were

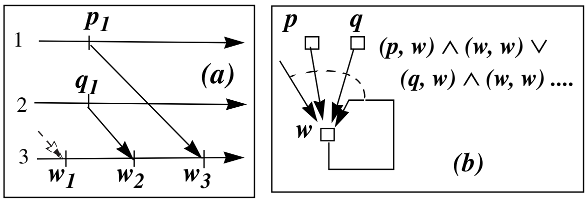
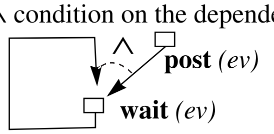


FIGURE 2. Occurrences of actions; *p*, *q* & w. *p* & *q* are post *(ev)* of process 1 & 2. w is wait *(ev)* of process 3.

caused by the dependence *(w, w)*.

Furthermore, the dependence information obtained from various instances of an action allows our debugger to infer the conditions that govern the action's execution. For example, the dependence information from w_2 and w_3 of Fig. 2(a) allows the debugger to infer that the condition "$((p, w) \wedge (w, w)) \vee ((q, w) \wedge (w, w))$" governs the execution of ***w***. The immediate predecessors of w_3 indicate that the condition "$(p, w) \wedge (w, w)$" must have initiated the third instance of ***w***. Similarly, immediate predecessors of w_2 indicate that the condition "$(q, w) \wedge (w, w)$" must have initiated the second instance of ***w***. The "$\vee$" operator appears because either of the two conditions can initiate the execution of ***w***. Now suppose that ***p***, ***q*** and ***w*** represented the synchronization primitives **post** (*ev*) and **wait** *(ev)*. Then, the condition brings out the information implicit in the semantics of the primitives.

Therefore, a user can represent M as conditions on the dependences of actions. Then, representing (**w** $\wedge$ **p**) precede **w** simply requires that we select the actions corresponding to **p** and **w**, connect them with the appropriate dependences, and specify an $\wedge$ condition. This is shown below by the $\wedge$ condition on the dependences of *post(ev)* and *wait(ev)*.



2.0 THE SPECIFIED BEHAVIOR

Our model decomposes the problem of debugging a concurrent program into two levels. A programmer debugs the concurrent state at the upper level, where the only important concerns are the relations on the set of computation actions; specified in the program, modeled in the expected behavior and observed in the execution behavior. As execution occurrences of computation actions are atomic, internal states of a computation action are not important at this level. These states only become important when the programmer moves to the lower level, inside the computation action, to debug its internal sequential text.

We represent the data dependences that force the computation actions to execute in a particular order by ordered pairs. If Σ_P denotes the set of computation actions, then:

Def. 2: *Data-flow dependences* are $F_P \subseteq \Sigma_P \times \Sigma_P$.

For instance, the dependence of a receipt of a message on its send, the dependence of a P of a semaphore on its V, or the dependence of a wait of a synchronization event on its post, represent such data-flow dependences. The write-read dependence on a shared synchronization variable, or on a message, forces the actions to execute in a particular order. Our motivation for representing the synchronization dependences as data-flow dependences comes from the language independence and machine independence goals of the CODE graphical programming environment [2], [15]. The data flow characterization of the synchronization and control-flow dependences in CODE allow the environment to support shared memory, as well as, distributed systems.

The set of ordered pairs F_P gives a graphical representation (Σ_P, F_P) whose nodes are the set of computation actions and whose arcs are the data-flow dependences. See Fig. 4(a). Intuitively, a computation action acts like a procedure whose input parameters are the input dependences and output parameters are the output dependences. It begins execution by obtaining a set of values from its input dependences. Then, it performs a sequential computation on this data. It ends its execution by putting a set of values on its output dependences.

Input dependences of an action ***a*** are given by the incoming arcs; $in(a) \equiv \{(x, a) \mid (x, a) \in F_P\}$. And, output dependences are given by the out-going arcs; $out(a) \equiv \{(a, x) \mid (a, x) \in F_P\}$. Conditions specified on the input dependences determine when to initiate the execution of a computation action, and conditions specified on the output dependences determine what follows its execution. The pre-condition that initiates the execution of a computation action is called an *input firing-rule*, and the post condition that follows its execution is called the *output firing-rule* [15].

Def. 3: An *input firing rule* $I_P(a)$ is a set of subsets of input dependences, i.e. $I_P(a) \subseteq 2^{in(a)}$. An *output firing rules* $O_P(a)$ is a set of subsets of output dependences; i.e. $O_P(a) \subseteq 2^{out(a)}$.

An input firing rule $I_P(a)$ is a condition in the disjunctive normal form (sum of products). Each element of $I_P(a)$ represents a disjunct, and is given by a subset of *in(a)*, input dependences of ***a***. The state of a data-flow dependence (x, a) can be represented by a string of values denoted by [x, a]. A computation action is ready for execution if the state of all the dependences of an element $\iota \in I(a)$ are non-empty strings i.e. $\forall\ (x, a) \in \iota$ :: $[x,a] \neq \varepsilon$. Then, input to a computation ***a*** is a set of suffix values detached from the state of dependences in ι.

On completing its computation, ***a*** will catenate a set of output values as prefixes to the state of all the dependences given in some element $o \in O(a)$.

3.0 THE OBSERVED BEHAVIOR

We trace the execution occurrences of computation actions and their orderings in P → E part of the debugging cycle. Fig. 4(b) shows this information.

Def. 4: *A computation event* is an execution occurrence of some *computation action*.

The set of computation events is denoted by V. Not all the computation actions of a program may occur in an execution. A subset of computation actions that do occur are, then, collected in Σ. Thus, $\Sigma \subseteq \Sigma_P$.

An action can occur multiple number of times. Subscripts in Fig. 4(b) denote the multiple occurrences of actions. Set V of events is, thus, a set of multiple occurrences of actions, or a "multiset" of occurrences of actions. The function μ: $V \to \Sigma$ maps each event of V to that action of Σ, of which it is an occurrence.

We support this mapping by associating an instance counter *u.i* with each *action* $u \in \Sigma_P$. The id-instance pair; *u* and *u.i*, provide a unique identifier for each event. We, therefore, denote the *i*-th occurrence of actions *u, v, w* ... as events u_i, v_i, w_i, ...

3.1 Causality of Data Flow Dependences

In order to record the orderings enforced by F_P, the debugger appends the unique identifier with the data shared through the data-flow dependences. Whenever an action *u* puts some data on its output dependence $(u, v) \in F_P$ in its *i*-th instance, it appends the identifier u_i to the data. Similarly, whenever an action *u* begins its *i*-th execution by removing some data from its input dependence $(v, u) \in F_P$, it detaches the identifier appended to the data, and puts the detached identifier in a predecessor list denoted by *u.i.P*. The list contains such pairs for all the predecessor events that have "caused" the *i*-th instance of *u*. The traces contain records of the execution instances of each event. The trace record of an event u_i contains the action id *u,* instance number *u.i,* and the predecessor list *u.i.P.* Also, see Table 1.

Def. 5: The orderings enforced by F_P are $<^F \equiv \{(u_i, v_j) \mid u_i, v_j \in V \wedge u_i \in v_j.P\}$.

The transitive closure of $<^F$ results in an irreflexive partial ordering that constitutes the *causal orderings* $<^c$. The orderings $<^F$ simply reflect the "causality" of data-flow dependences.

Lemma 1: $u_i <^F v_j \Rightarrow (u, v) \in F_P$.

Consequently, if $u_i <^F v_j$, then there is some data shared between u_i and v_j, namely, the state of the data-flow dependence $(u, v) \in F_P$. The representation of the state of a data-flow dependence in § 2.0 by a string of values (or an infinite FIFO buffer) takes into account this dependence. Moreover, it allows us to model the general cases of the send and receive of messages in distributed systems, and the data-flow dependences of the graphical/visual languages like CODE [2],[15]. But, the representation may create problems in modeling the synchronization primitives of shared memory systems. However, as seen in § 3.3, concurrent state does not depend upon the internal representation of the states of the data-flow dependences. It only depends upon the event orderings. Thus, we can represent the state of a synchronization dependence, with a string (or a buffer) of length one. It will denote the data dependence due to the shared synchronization variable (whose only permitted values are set or reset). The debugger records the causal orderings by appending and detaching the identifier to the shared synchronization variable.

3.2 Execution History Pomsets

As seen above, actions can occur multiple number of times as events, i.e. the set V of events is related to the set Σ of actions through the function μ: $V \to \Sigma$. This effectively turns the poset $(V, <^c)$ into a pomset $(\Sigma, V, <^c, \mu)$ [19]. A POMSET is a Partially Ordered MultiSET of occurrences of **actions**, in much the same way as a **string** is a TOMSET; a Totally Ordered MultiSET of occurrences of **alphabets**. The pomset $(\Sigma, V, <^c, \mu)$ is called a causal pomset because $<^c$ are the causal orderings. It is instrumental in unifying our model because its expression of the concurrency properties is independent of the way time or events are modeled in a system [5].

A pictorial representation of the causal pomset is an *execution history display.* We can now explain why execution history displays are so helpful in debugging. They display the causal orderings of events. These orderings allow a programmer to determine the conditions that initiated and followed each execution instance of an action. From Lemma 1, an immediate predecessor of $\boldsymbol{u_i}$ must map to an input dependence of ***u***; and from Def. 3, an element of the input firing rule of ***u*** is a subset of input dependences. Hence, immediate predecessors of an event $\boldsymbol{u_i}$ inform the programmer about that element of the input firing-rule that initiated the ***i***-th instance of ***u***. Similarly, immediate successors of an event $\boldsymbol{u_i}$ inform the programmer about that element of the output firing-rule that determined the condition following the ***i***-th execution instance of ***u***. If ${}^{\bullet}u_i \equiv \{(v, u) \mid v_j <^F u_i\}$ and $u_i^{\bullet} \equiv \{(u, v) \mid u_i <^F v_j\}$, then:

Def. 6: A causal pomset $(\Sigma, V, <^c, \mu)$ is *compatible* with the firing rules iff $\forall u_i \in V :: {}^{\bullet}u_i \in I_P(u) \wedge u_i^{\bullet} \in O(u)$.

This shows the compatibility of the immediate orderings of a given event with the firing rules specified on the immediate dependences of its corresponding action. In § 4.0, we extend this compatibility of immediate orderings with the immediate dependences to the compatibility of transitive orderings with the transitive dependences. This provides a framework for representing and checking the expected behavior.

3.3 Concurrent Execution State

There is non-determinism associated with the choices of the elements given in the input and output firing rules. An action can non-deterministically select different elements of a firing rule. In Fig. 2, the second instance of action w can non-deterministically select any element from its *input firing rule*. During execution replay, a record of the causal orderings informs our debugger to select the right element of the firing rules for each execution instance of an action. Thus, it reconstructs the states of the previous execution. Using $<^c$ and following [13], we find a notion of concurrent state:

Def. 7: A *concurrent state* of an execution $(\Sigma, V, <^c, \mu)$ is a consistent cut-set of the poset $(V, <^c)$. A set $C \subseteq V$ is a consistent cut-set iff $e \in C \wedge e' <^c e \Rightarrow e' \in C$.

This definition is independent of the local state of an action. It is, also, independent of the contents of the messages exchanged by the actions. It only depends upon the order of events. Hence, distributed systems often reduce their roll-back and recovery overhead by only recording the event orderings, and not the content of messages or the checkpoints of the local states. Therefore, our execution replay facility exploits this definition to reduce the recording overhead (§ 5.1).

3.4 Animation

Animation provides an instantaneous view of the execution history. It is simply the process of displaying Def. 6 on an animation structure while traversing the execution pomset. An *animation structure* shows the assignment of actions to the executable units (processes/threads) of the execution environment. It can be considered as an elaborated form of the program structure. Fig. 3(c) shows such an animation structure where action w of Fig. 3(a) replicates (or elaborates) into three actions; w^1, w^2, and w^3 running on different processes. We represent the run-time assignment of actions Σ_P to the executable units by the set $\Sigma \subseteq \Sigma_P \times N$; where, w^i, $w^j \in \Sigma$ implies that w is assigned to different executable units whose logical ids are i and j. The actual identity of an executable unit is not important. The superscripts i, j are only for distinguishing between the multiple copies.

During animation, the debugger traverses the execution pomset. On encountering the i-th event instance of an action u^k, it highlights in the animation structure those input dependences of the action that correspond to the immediate predecessors of its i-th event instance. It, then, highlights the action u^k. Then, it highlights those output dependences of the action that correspond to the immediate successors of its i-th event instance.

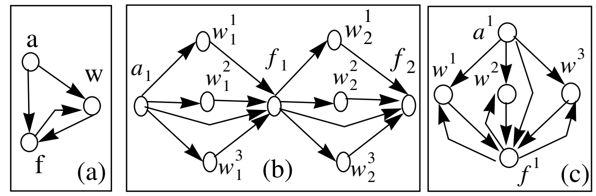


FIGURE 3. (a) *Program structure*, (b) *pomset execution*, and (c) elaborated *animation structure*.

We can automatically generate an animation structure from the execution pomset. Note that the structure shown in Fig. 3(c) is obtained by folding all the subsequent instances of actions in Fig. 3(b) to their first occurrences. This fulfills the requirement of a strong coupling between animation and execution history [14].

4.0 THE EXPECTED BEHAVIOR

In the P ⟶ M part of the cycle, a user represents the expected behavior by selecting some "interesting" actions Σ_M from Σ_P and, then, representing the expectations as conditions I_M, O_M on their dependences F_M.

4.1 Representing Expected Behavior

A *dbx* debugger is closely coupled with the program because it compels the programmer to use only those objects that already exist in the program; e.g. it would not allow a user to specify a non-existent print variable. Taking cue from *dbx*, we closely couple our checker with the program and only allow the user to work with those objects that already exist in the program. This is in contrast to the existing checkers that can not verify if a user has supplied a non-existent order of events in the expected behavior.

Note that if a user expects that events u_i and v_j will be ordered in the execution, then their actions u and v, must exhibit a (transitive) data-flow dependence in the program. That is, $u_i <^c v_j \Rightarrow (u, v) \in F_P{}^*$, where $F_P{}^*$ is a transitive closure of F_P. This also follows from Lemma 1. For instance, $m_1 <^c c_1$ in Fig. 4(b) corresponds to the transitive *data-flow dependence* between m and c of Fig. 4(a); both are shown by dotted lines. Thus, any pattern that is expected in an execution, must be the unrolling of a pattern already present in the program structure.

Thus, a user starts specifying expected behavior by selecting a subset Σ_M of interesting actions from the program; $\Sigma_M \subseteq \Sigma_P$. Fig. 4(c) shows a selection of such actions. The user can then specify a dependence

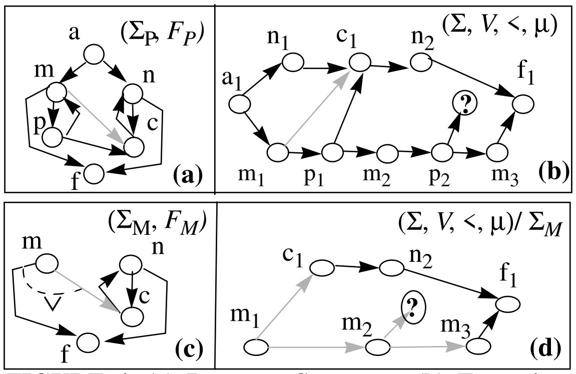


FIGURE 4. (a) Program Structure. (b) Execution. (c) Expected Behavior. (d) Restricted execution.

between the selected actions only if it corresponds to some dependence of F_P*. Some of such selected dependences are shown as F_M in Fig. 4(c).

Def. 8: F_M are the selected dependences from the transitive *data-flow dependences* F_P^* restricted to interesting actions i.e. $F_M \subseteq F_P^* / \Sigma_M$, where $\Sigma_M \subseteq \Sigma_P$.

An observed ordering like *$(m_2, ?)$* in Fig. 4(d), that can not be mapped to a data-flow dependence from F_M is, therefore, symptomatic of a bug!

In Fig. 4(c), the structure built on Σ_M is the specification of the expected behavior. Akin to the conditions in a *dbx* command like **"when at** stmt **if** condition," a user can, then, provide firing rules I_M and O_M to further restrict the instances of "interesting" actions. Note that firing rules are conditions about the concurrent state, whereas *dbx* conditions are about the sequential state. Suppose the user specifies an "∨" *output firing rule* for action ***m*** in Fig. 4(c). Then, the checker can filter out instances $\boldsymbol{m_1}$ and $\boldsymbol{m_3}$ because they subscribe to the "∨" rule. But, will raise an exception for $\boldsymbol{m_2}$ as it does not subscribe to the "∨" rule.

4.2 Recognizing the Expected Behavior

In *dbx*, a directive like "**when at** stmt **. . . .**" informs the debugger to make necessary preparations for the specified statement. It also informs it to ignore the rest of statements. Similarly, the selection of Σ_M from Σ_P informs the debugger to specially prepare for "interesting" actions Σ_M, and to safely ignore the "uninteresting" actions Σ_P - Σ_M. Then, the debugger can filter out the uninteresting events and can restrict the execution to instances of Σ_M.

The restricted pomset $(\Sigma, V, <^c, \mu)/\Sigma_M$ of Fig. 4(d) only contains the interesting events and their mutual orderings. The debugger establishes the orderings among interesting events by exploiting the fact that instances of interesting actions can only be ordered if there exists a mutual (transitive) dependence. In Fig. 4(d), events $\boldsymbol{m_1}$ and $\boldsymbol{c_1}$ are only ordered because of the transitive dependence that exists between actions ***m*** and ***c***. Note in Fig. 4(a) that the dependence goes through an intervening uninteresting action ***p***. Our debugger, therefore, establishes the orderings between instances of ***m*** and ***c***, by asking the uninteresting action ***p*** to relay the causality information that arrived from its predecessors, forward to its successors.

Unlike interesting actions, uninteresting actions do not trace their execution instances. Instead of sending the identifier of their current instance to their successors, they simply relay forward their predecessor lists. See Table 1. These lists keep getting relayed forward by the intervening uninteresting events until they land in the predecessor lists of the interesting events. Only then they are traced. Predecessor lists of interesting events, therefore, only contain the identifiers of their causally preceding interesting events. Execution is thus filtered to $(\Sigma, V, <^c, \mu) / \Sigma_M$.

Table 1: Monitoring and tracing of actions.

Monitored Occurrences	Interesting Actions $u \in \Sigma_M$	Uninteresting Actions $u \in \Sigma_P - \Sigma_M$
u **sends** to *v*	**append**(*msg: u, u.i*);	**append** (*msg: u.i.P*);
u **receives** *msg*	*u.i.P* ∪**detach**(*msg*);	*u.i.P* ∪ **detach**(*msg*);
u **executed**	**trace** *(u, u.i, u.i.P)*; *u.i := u.i + 1;*	*u.i := u.i + 1;*

The structural information of M and the fact that the causal pomset is restricted to interesting actions, greatly simplifies recognition of the expected behavior. The debugger traverses the partial order, and tries to check if Lemma 1 and Def. 6 also hold for M. It checks whether each immediate successor v_j of a given event u_i corresponds to some successor *v* of the action *u* in the dependences F_M. Additionally, the debugger checks whether the immediate predecessors and successors of an event u_i satisfy the input and output firing rules I_M and O_M for *u*. If an ordering $u_i <^F v_j$ fails to correspond to some dependence $(u, v) \in F_M$, or the immediate predecessors

${}^{\bullet}u_i$ or successors $u_i^{\bullet}$ fail to meet the expected conditions $I_M(u)$ or $O_M(u)$, then an error has been recognized. This happens for the unexpected orderings of m_2 in Fig. 4(d).

Def. 9: Event u_i is in *error* if $u_i^{\bullet} \notin O_M(u) \vee {}^{\bullet}u_i \notin I_M(u)$.

Thus, concurrent debugging is the process of following the unexpected orderings given by the erroneous events, in the direction of causality.

5.0 SHARED DATA DEPENDENCES

Unlike dependences F_P that force an ordering on the execution of actions, shared data dependences do not impose any particular orderings. We, therefore, model a shared data object by the set of actions that share it. The set of those objects is denoted by S.

Def. 10: The set of *shared data dependences* is $S \subseteq 2^{\Sigma_P}$. A *shared data dependence* is a set D of *computation actions*, $D \subseteq \Sigma_P$ (or $D \in S$).

The actions that participate in a shared data dependence D are classified into disjoint sets of *readers* and *writers*. *Readers* of D are $\rho.D$ and *writers* are $\omega.D$; where $\rho.D \cup \omega.D \equiv D$, $\rho.D \cap \omega.D \equiv \phi$. In Fig. 5(a), shared data dependence $D = \{a, r, w\}$. For example, we may have the set of *readers*, $\rho.D = \{r\}$ and set of writers, $\omega.D = \{a, w\}$.

5.1 Execution Replay

The goal of an execution replay facility is to record enough information about the non-deterministic choices made by the events of an execution. During replay, the events can, then, be forced to make the previous choices. A record of causal orderings $<^F$ is sufficient to overcome the non-determinism associated with the choices of elements in a firing rule. But, there is another source of non-determinism. This is associated with shared data dependences that do not force the actions to execute in any particular order. As accesses to shared objects can take place in any order, the debugger must record the non-deterministic order of accesses to shared objects so that during replay the accesses can be forced to occur in the previously recorded order.

We record the order of accesses to shared objects with the same mechanism that we use in recording the $<^F$orderings (§ 3.1). But before we can do that, we need a protocol to ensure a valid serialization on the accesses to shared objects like the CREW (concurrent read exclusive write) protocol [10]. The protocol disallows simultaneous write-access to a shared object with other accesses. The debugger ensures a deterministic replay by implementing the protocol. Without the implementation, simultaneous accesses to a shared object can hinder a deterministic replay by corrupting the object and giving unpredictable results. Note that members u

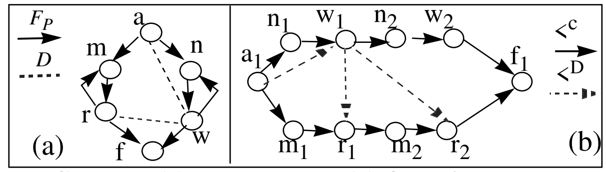


FIGURE 5. (a) Dependences. (b) Orderings.

and v of a shared data dependence have a valid serialization if for every instance u_i of a write-access and every instance v_j of another access, either u_i occurs before v_j, or v_j occurs before u_i

Unlike [10] that uses versions of shared objects to record the orderings, we use our simpler mechanism for recording the order $<^D$ of accesses to a shared object $D \in S$. Each shared object has the identifier of the last instance of its write-access appended to it. Whenever an action accesses a shared object, it reads the identifier appended to the object, and places the identifier in the predecessor slot reserved for that object in its trace event record. A writer, in addition to the above, replaces the identifier appended to the object with the identifier of its present instance.

In addition to the predecessor list $u_i.P$ for the data-flow dependences (§ 3.1), the trace record for each event now requires another predecessor list $u_i.S$ for shared data dependences. The list has a slot $u_i.S[D]$ for each shared data dependence D in which a given action participates. Then, the identifier in the slot for D in the predecessor list of an event record, determines the causal orderings $<^D$. There is, thus, an ordering relation $<^D$ for each $D \in S$ generated like $<^F$. The debugger, then, ensures the replay by forcing the events to occur in the pre-recorded order.

To simplify the following discussion, we assume that there is "one" shared data dependence $D \in S$. We now consider the strict partial orders due to $<^F$and $<^D$.

5.2 Race Detection

Simultaneous accesses (with at least one write) to a shared object can race with each other and can corrupt the shared data with unpredictable results. We detect races by identifying those pairs of events whose accesses to a shared object included at least one write and whose orderings were only due to the debugger's enforcement of the serialization protocol. For e.g., dotted arcs of Fig. 5(b) show the $<^D$ orderings for events $\boldsymbol{w_1}$ and $\boldsymbol{r_1}$ (and $\boldsymbol{w_1}$ and $\boldsymbol{r_2}$) that were forced by the debugger's serialization protocol. The events are otherwise unordered under $<^F$. Without the debugger's protocol, these $<^D$ orderings may not exist. Then, $\boldsymbol{w_1}$ and $\boldsymbol{r_1}$ (or $\boldsymbol{w_1}$, $\boldsymbol{r_2}$) can execute simultaneously with unpredictable results. Thus, all $<^D$ orderings observed under the serialization protocol, should be supported by $<^F$ as, for instance, the ordering between $\boldsymbol{a_1}$ and $\boldsymbol{w_1}$.

Although this technique detects the possibility of race for events w_1 and r_1 (and for events w_1 and r_2) of Fig. 5(b), it does not detect the possibility of race between events w_2 and r_2. To detect such races, we note that these events are unordered by both $<^D$ and $<^F$. Thus, our debugger will also signal the races for all those pairs of events that are unordered by $<^D$ and $<^F$, and that access a shared object with at least one write.

Implementing a serialization protocol, and recording the $<^D$ orderings of accesses to a shared object, may seem unnecessary for detecting races. It may appear simpler to report races for pairs of events that are unordered under $<^F$. However, as explained in [16], this can result in reports of spurious races that are infeasible and could never occur. Our debugger's implementation of the serialization protocol is instrumental in eliminating the spurious artifacts that can result from the use of shared objects that were corrupted by an earlier race. Furthermore, the record of $<^D$ helps in improving the accuracy of detected races by identifying other spurious races. Note that race detection in our model does not require any extra overhead. The record of $<^D$ and $<^F$ already exists for supporting execution replay.

6.0 CONCLUSIONS

The unified model of concurrent debugging presented in this paper covers all the parts of the debugging cycle. This overcomes the incompatibility of existing facilities, and allows our debugger to support different debugging facilities like *execution replay*, *race detection*, *assertion/model checking*, *execution history displays*, and *animation*. We, thus, show that the benefits of modeling the whole cycle are greater than a simple sum of its parts.

Our model uses the set of computation actions and the "causality" of their dependences to simplify the complexity of concurrent debugging. We show that it is easier for a debugger to record the orderings when events are considered to be the occurrence instances of actions. We, also, show that the debugger can obtain much more information by recognizing the underlying dependences of the observed orderings.

Our use of a program oriented approach for checking the model of expected behavior saves the user extra programming effort. It, also, simplifies for the debugger, filtering and recognition of the expected behavior.

Our model proposes a solution for the following needs highlighted in the panel discussions of [11]. It provides a theoretical framework for defining an ***error*** and explaining the process of concurrent debugging (J. Wilden)[1]. The framework explains precisely when and why a particular facility is needed (P. Bates). It eases implementation by separating the monitoring and presentation concerns. It is not specific to a particular language, or a particular system. It proposes the abstraction of *computation actions* to fulfill the need for a lower limit for the granularity of data collection (A. Tilberg). Such an abstraction is usually available in a graphical environment like CODE [2], [15]. However, as explained in the Appendix, some static analysis may be necessary to obtain the *computation actions* from a textual representation of a concurrent program.

References

[1] P. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior," *ACM SIGPLAN Notices*, 24(1), (Jan '89), pp. 11-22.

[2] J. C. Browne, M. Azam, and S. Sobek, "A Unified Approach to Parallel Programming", *IEEE Software,* (Jul '89).

[3] D. Callahan and J. Subhlok, "Static Analysis of Low-Level Synchronizations," *ACM SIGPLAN Notices*, 24(1), (Jan '89)

[4] R. J. Fowler, T. J. LeBlanc and J. M. Mellor-Crummy, "An Integrated Approach to Parallel Program Debugging and Performance Analysis," *ACM SIGPLAN Notices*, 24(1), (Jan '89), pp. 163-73.

[5] H. Gaifman, "Modeling Concurrency by Partial Orders and Nonlinear Transition Systems," *Lecture Notes on Comp. Science #354*, (May '88), pp. 467–88.

[6] A. A. Hough and J. E. Cuny, "Perspective views: A Technique for Enhancing Parallel Program Visualization," *ICPP*, (Aug '90), pp. II.124–II.132.

[7] W. Hseush and G. E. Kaiser, "Modeling Concurrency in Parallel Debugging," *ACM Symp. on Princples & Practice of Parallel Prog.*, (March '90), pp. 11–20.

[8] D. P. Helmbold, C. E. McDowell, and J. Wang, "Analyzing traces with anonymous synchronization," *ICPP*, (Aug. '90), pp. II.70–II.77.

[9] A. A. Hough, "Debugging Parallel programs Using Abstract Visualizations," TR 91:53, CS Department, University of Massachusetts at Amherst, (Sep '91).

[10] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. on Computers*, C36 # 4, (Apr '87), pp. 471-81.

[11] T. J. Leblanc and B. P. Miller, Ed.s, "Workshop Summary; What we have learned and where we go from here?" *ACM SIGPLAN Notices, 24(1)*, (Jan, 89), pp. ix–xxii.

[12] T. J. LeBlanc, J. Mellor-Crummey & R. J. Fowler, "Analyzing Parallel Executions with Multiple Views," *J. of Paral. & Dist. Comp.* #9, (Jun '90), pp. 203-17.

1. Parenthesized names indicate the person who highlighted the need in the panel discussion of [11]

[13] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, (1989), pp. 215–26.

[14] C. E. McDowell, D. P. Helmbold, "Debugging of Concurrent Programs," *ACM Computing Surveys*, 21(4), (Dec '89), pp. 593–622.

[15] P. Newton and J. C. Browne, "The Code 2.0 Graphical Programming Environment," *Supercomputing* '92 (Jul '92).

[16] R. H. Netzer and B. P. Miller, "Improving Accuracy of Data Race detection*," ACM SIGPLAN Notices 26(7)*, (Jul '91), pp. 133-44.

[17] R. H. B. Netzer and B. P. Miller, "On the Complexity of Event Ordering for Shared Memory Programs," *ICPP*, (Aug '90), pp. II.93–II.97.

[18] R. H. Netzer and B. P. Miller, "What are Race Conditions? Some Issues and Formalism," TR91-1014, CS Dept. Univ. of Wisconsin (Mar '91).

[19] V. Pratt, "Modeling Concurrency with Partial Orders," *Internaltional Journal of Parallel Programming*, 15(1), (1986), pp. 33–71.

[20] C. M. Pancake and S. Utter, "Models for Visualization in Parallel Debuggers," *Supercomputing '89*, (Nov '89) pp. 627–36.

[21] S. Utter and C. M. Pancake, "A Bibliography of Parallel Debugging Tools," *ACM SIGPLAN Notices*, 24(10), (1989), pp. 24-42.

Appendix: Computation Actions in a Textual Representation of a Program

A textual program contains three types of statements; blocking synchronization*,* signal synchronization and non-synchronization statements. From such a text, *static analysis* routinely extracts a synchronization-control-flow graph that contains three types of nodes and two types of arcs [14], [3]. Nodes represent blocking synchronization, signal synchronization, and control decision statements. While arcs represent inter-process synchronization dependences and intra-process control-flow dependences.

Fig. 6(a), (b), and Fig. 7(a), (b) show the synchronization control flow graphs obtained from a textual program of a PPL like extension of C. Intra-process control arcs labeled by ***a***, ***w*** and ***f*** correspond to the sequential text containing non-synchronization statements. The abstraction of computation actions, shown by the dotted ovals, permanently associates the synchronization statements with the sequential texts. It permanently associates a blocking synchronization with the sequential text that follows it, and associates a signal synchronization with the sequential text that precedes it. Thus, blocking synchronizations **wait** *(ev_i)* is associated with the text ***w*** that follows it, and blocking synchronization **c_wait**

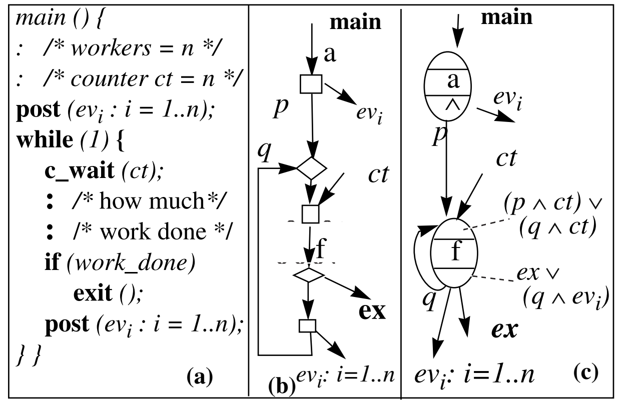


FIGURE 6. (a) *main* starts *n* workers, and waits on a counter *ct*. On completion, each *worker* reports by incrementing *ct*. When count reaches *n*, *main* wakes up. If there is need for more work then, it signals the *workers* again, else it exits. (b) *Synchronization-control-flow* graph. (c) Actions *a* and *f*.

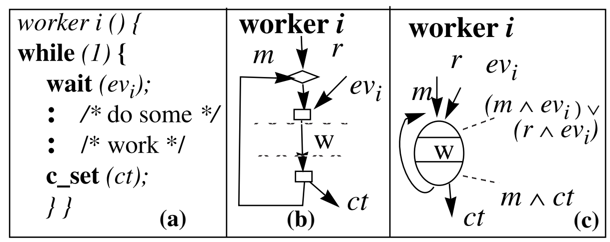


FIGURE 7. (a) Text. (b) *Graph*; *m*, *r* are control dependences; *ct* and ev_i are synchronization dependences. (c) Action *w* ; $(m \wedge ev_i) \vee (r \wedge ev_i)$ is *input-rule*, $m \wedge ct$ is the *output-rule*, and *w* is the sequential text.

(ct) is associated with the text ***f*** that follows it. Also, *signal* synchronizations **c_set**(*ct*) is associated with the text ***w*** that precedes it, and signal synchronization **post** *(ev_i : i = 1..n)* is associated with texts ***a*** and ***f*** that precede it.

Def. 11: A *computation action* is a block of a flow graph that:

1. may contain internal control flow provided the internal control structures (loop, if-then-else, etc.) do not contain any synchronization statement;
2. it may begin with a *blocking* synchronization, that must be the first statement of the block; and
3. it may end with a *signal* synchronization, that must be the last statement of the block.
4. it may contain more than one *blocking (signal)* synchronization provided all are together at the start (end) of the block with no other intervening statement.

After the synchronization statements have been associated with their bordering sequential texts, we are left with the control flow decision nodes. The abstraction of firing rules subsumes these decision nodes as shown in