11-26-2001

Shared Variables Interaction Diagrams

Rajeev Alur
University of Pennsylvania, alur@cis.upenn.edu

Radu Grosu
State University of New York at Stony Brook

Shared Variables Interaction Diagrams

Abstract
Scenario-based specifications offer an intuitive and visual way of describing design requirements of distributed software systems. For the communication paradigm based on messages, *message sequence charts* (MSC) offer a standardized and formal notation amenable to formal analysis. In this paper, we define *shared variables interaction diagrams* (SVID) as the counterpart of MSCs when processes communicate via shared variables. After formally defining SVIDs, we develop an intuitive as well as formal definition of refinement for SVIDs. This notion provides a basis for systematically adding details to SVID requirements.

Shared Variables Interaction Diagrams

Rajeev Alur
Department of Computer and Information Science
University of Pennsylvania
alur@cis.upenn.edu

Radu Grosu
Department of Computer Science
State University of New York at Stony Brook
grosu@cs.sunysb.edu

Abstract

Scenario-based specifications offer an intuitive and visual way of describing design requirements of distributed software systems. For the communication paradigm based on messages, message sequence charts *(MSC) offer a standardized and formal notation amenable to formal analysis. In this paper, we define* shared variables interaction diagrams *(SVID) as the counterpart of MSCs when processes communicate via shared variables. After formally defining SVIDs, we develop an intuitive as well as formal definition of refinement for SVIDs. This notion provides a basis for systematically adding details to SVID requirements.*

1. Introduction

Message Sequence Charts (MSCs) are a commonly used visual description of design requirements for concurrent systems such as telecommunications software [19], and have been incorporated into software design notations such as UML [6]. On one hand, the clear graphical layout of an MSC immediately gives an intuitive understanding of the intended system behavior, and on the other, the notation has been standardized (ITU standard Z.120) with precise semantics, and hence, can be subjected to analysis. This has already motivated the development of algorithms for a variety of analyses including detecting race conditions and timing conflicts [3], pattern matching [16], detecting nonlocal choice [7], and model checking [4], and tools such as uBET [11] and MESA [5].

An MSC depicts the desired exchange of messages among communicating entities in distributed software systems. An alternative paradigm for communication in distributed systems involves shared variables. Communication and synchronization via shared objects provides a higher level of abstraction, and is supported by many modern concurrent programming languages. In this paper, we propose *shared variables interaction diagrams* (SVID) as a formal and visual notation for describing scenarios in the shared variables paradigm. Textbooks on concurrent programming (e.g., [14, 17]) contain many pictures describing the interactions of processes communicating by shared variables, and similar scenarios arise in diverse areas such as transaction processing in concurrent databases (c.f. [18]) and consistency in shared-memory multiprocessors [13].

In our definition of an SVID, an action corresponds to, possibly multiple, reading/writing of shared variables. The actions of one process are visually ordered. The causal dependence among actions of different processes is illustrated by arrows: an arrow from an action a of process p to an action b of process q means that q reads a value that was written by p. If the variable involved in this communication is write-shared, then there is an implicit additional constraint that between these two actions a and b, there is no intervening action that writes to this variable. Checking whether all such implicit constraints are consistent with one another can be computationally hard, and is shown to be NP-complete in general. We also establish that the problem can be solved in linear-time when all the variables are read-shared but write-exclusive. Benefits of SVIDs are the same as that of MSCs: they give an intuitive and visual understanding of interactions among processes in a single execution, and have a formal semantics. The execution of a concurrent program can also be depicted by a linear trace involving actions of all processes, but a single SVID captures many such executions succinctly, and makes causal dependencies explicit.

An appealing notion for systematic hierarchical development of specifications or models involves refinement (this is present in all concurrency formalisms [20]). The definition of SVIDs, and the underlying shared-variables paradigm, suggests many natural ways of refining SVIDs. We identify different ways of visually adding more details to an SVID: by moving arrows depicting dependencies, by introducing new variables, actions, and/or new arrows, by splitting composite actions, and by splitting processes into subprocesses. All these cases are captured by our formal definition of refinement for SVIDs. The definition requires existence of a mapping of implementation actions to specification actions consistent with the dependencies. We show the problem of

0-7695-1426-X/01 $17.00 © 2001 IEEE

checking refinement to be NP-complete. When the communication is point-to-point, that is, each variable has a single writer and a single reader, the problem can be solved in polynomial time.

The remaining paper is organized as follows. Section 2 introduces the definition of SVIDs. Section 3 defines the notion of refinement for SVIDs. Section 4 compares SVIDs to related formalisms, and in particular to message sequence charts (MSCs).

2. Shared Variables Interaction Diagrams

Peterson's mutual exclusion protocol. In order to illustrate the use and utility of *shared variables interaction diagrams (SVID)* let us consider the Peterson's mutual exclusion protocol for two asynchronous processes p_1 and p_2. The protocol makes sure that p_1 and p_2 never simultaneously reach their critical sections and that each may eventually enter its critical section provided it desires to do so.

To achieve the desired synchronization among p_1 and p_2 the protocol uses three variables. The first variable f_1 is a boolean variable (or flag), that when set, signals that p_1 desires to enter its critical section. It is writable only by p_1 but it can be read by p_2. The second variable, the flag f_2, plays the same role for p_2 as f_1 does for p_1. Finally, the variable t (turn) is used to resolve the conflict when both processes try to simultaneously enter their critical sections. The variable is written and read by both processes and ranges over the set of process identifiers, i.e., $\{1, 2\}$ in the binary case.

Exemplary SVIDs for the protocol. In Figure 1 we show four typical scenarios for the Peterson's protocol as *basic* SVIDs. They intuitively capture the synchronization (communication) patterns between the processes p_1 and p_2. The SVIDs M_1 and M_3 describe the situations where only p_1 or only p_2 requests the critical section. The SVIDs M_2 and M_4 show how the tie is resolved when both processes p_1 and p_2 request the critical section. These scenarios can be used for understanding the behavior of the protocol, or they can be used as a specification for designing the protocol. Clearly, there are many more scenarios that are possible, and the specified ones serve only as a guide.

As with message sequence charts (MSCs), vertical lines correspond to processes. However, in contrast to MSCs, the processes do not communicate with each other via messages. Instead, they communicate via *shared variables*. The way a process updates its variables is given textually in the left compartment of rectangular boxes (or vertices). The variables needed (or read) by the update operations in a box are given in the right compartment of the same box. Variables and updating operations inside a box are not ordered. They may be regarded as being performed simultaneously, i.e., they define an *atomic action*.

The inter-process communication via shared variables is depicted by arrows (or edges) between boxes. An arrow pointing from a box b_1 to a box b_2 indicates that the value of the variables updated by b_1 is read (or checked) by b_2. For instance, in Figure 1, in M_2, the arrow from b_{12} to b_{24} specifies that the value of f_1 read by p_2 in box b_{24} is the value written to f_1 by p_1 in box b_{12}. Thus, the arrows establish a causal order between actions of different processes: the action of p_2 corresponding to b_{24} must happen *after* the action of p_1 corresponding to b_{12}. For write-shared variables, the arrows establish an additional causal dependence between reads and writes, namely, not only the read should happen after the write, but in addition, there should be no intervening write to the shared variable. For instance, in M_2, the arrow from box b_{22} to b_{13} says that the value of t read by p_1 in box b_{13} is the value written to t by p_2 in box b_{22}. This means that the action of p_1 corresponding to b_{13} happens after the action of p_2 corresponding to b_{22}, and between these two actions there is no action involving writing to the shared variable t. In this case, this enforces an implicit causal dependence between box b_{12} and b_{22} (i.e, p_2 writes to t after p_1 in this scenario).

To simplify the notation we adopt the (usual) convention that, write exclusive variables of a process preserve their values if not explicitly updated. Hence, a box b contained by a process p exports not only the variables explicitly updated by b but also the write exclusive variables of p that are not updated by b. This allows the use of empty boxes (not explicitly drawn) as sources for arrows. Similarly to MSCs the arrows may be labeled. However, the arrow labels are not messages. They are conditions over variables that have to hold in order to perform the update operation at the head of the arrow.

The vertical process lines define a top down linear order among the update operations (boxes) of the same process (like in MSCs time flows top/down). Hence, vertical lines describe local synchronization whereas arrows describe global synchronization.

To improve readability we use, as with MSCs, *conditions*. They are drawn inside hexagonal boxes. Intuitively, a condition is an update of the program counter variable associated with the relevant process.

In the following we define SVIDs in a formal way. To simplify the definition we do not consider arrow labels. They might be understood simply as comments that reflect the most recent value of the associated variables. Similarly, we do not include the conditions in the formal definition.

Definition 1 (Shared variables interaction diagram) *A* shared variables interaction diagram M *consists of the following components:*

Processes. *A finite set P of processes. Each process $p \in P$ has an associated (finite) set $p.X$ of (typed)*

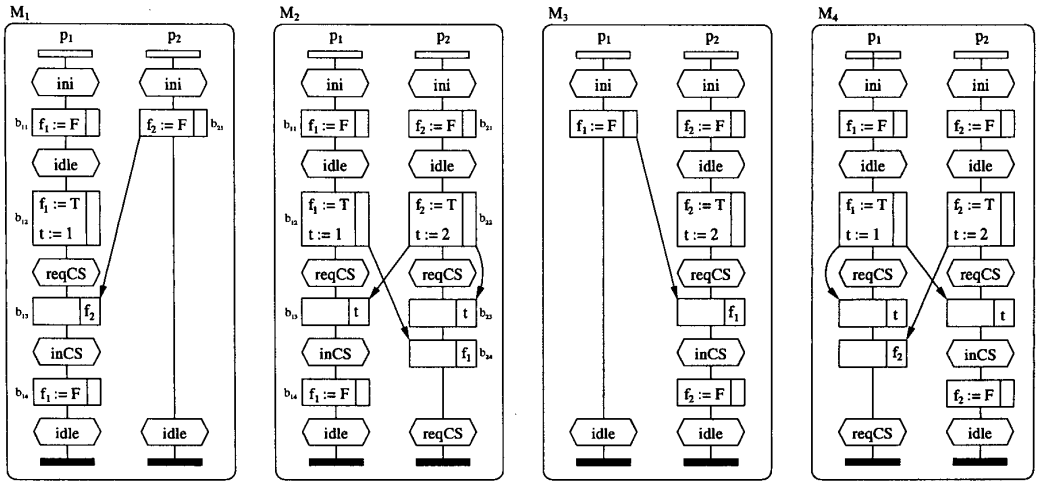

Figure 1. Basic SVIDs for the Peterson protocol

variables. *The variables in $p.X$ are classified into read variables $p.X_r$, write-shared variables $p.X_s$ and write-exclusive variables $p.X_e$. We denote by $p.X_w = p.X_s \cup p.X_e$ the set of write variables of p and by $X = \cup_{p \in P} p.X$ the set of all variables of M. It is required that the write-exclusive variables of different process are disjoint: for $p \neq q$, $p.X_e \cap q.X_e = \emptyset$, and every variable is written by some process: $M.X = \cup_{p \in P} p.X_w$.*

Vertices. *A finite set V of vertices. Each vertex v has an associated process $v.p$. The set of vertices belonging to a process p is denoted by $p.V$. This set is linearly ordered by the (top down) relation $<_p$. We denote by $<$ the partial order $\cup_{p \in P} <_p$ obtained by taking the union of these linear orders.*

Update boxes. *Each vertex v has an associated set $v.X_r$ of read variables, a set $v.X_w$ of write variables and a state $v.s$. The set $v.X_r$ is the set of variables read in order to perform the update, and it is required that $v.X_r \subseteq v.p.X_r$. The set $v.X_w$ is the set of variables explicitly updated, and it is required that $v.X_w \subseteq v.p.X_w$. Finally, the state $v.s$ corresponds to the update and it is a mapping that associates each variable $x \in v.X_w$ to a value $a \in T_x$ where T_x is the type associated to x* [1].

Read edges. *A finite set $E \subset V \times V$ of edges denoting the read dependencies. For each vertex v the set $v.X_r$ of read variables of v is included in the set of write and inherited variables of the immediate predecessors, i.e., $v.X_r \subseteq \cup_{(u,v) \in E}(u.X_w \cup u.p.X_e)$.* □

[1] The specific state associated with a vertex is not of importance for subsequent results. The definition can be made more general to allow more abstract states such as predicates, or to omit the state altogether.

We say that a SVID M is consistent if there is a linearization of the updates in M that respects the linear vertical (process) orders and the partial horizontal (arrows) order. In addition, the horizontal order should reflect most recent updates (see definition below). For example, in Figure 1, M_1 is a consistent SVID with three linearizations $b_{21}b_{11}b_{12}b_{13}b_{14}$, $b_{11}b_{21}b_{12}b_{13}b_{14}$ and $b_{11}b_{12}b_{21}b_{13}b_{14}$.

Definition 2 (Consistent SVID) *An SVID M is called* consistent *if there is a linear order $v_1 v_2 \ldots v_n$ of all vertices in $M.V$ such that (1) it is consistent with $<$, (2) it is consistent with E, and (3) if x is a variable and $e = (u, v)$ is an edge with $x \in v.X_r$ and $x \in u.X_w$ then there is no vertex w between u and v with $x \in w.X_w$.* □

The consistency requirement makes sure that the value assigned by a process to a variable in an update box at the tail of an arrow is also the value considered by the other process in the box at the head of the arrow. This condition is new when compared with MSCs and it is necessary because both the process containing the box at the head of an arrow or another process if the variable is write shared might have written the variable too.

Note that if a box b_1 reads a variable x, then the definition of an SVID requires that it has an incoming edge from some box b_2 that writes to x or inherits x. Multiple incoming edges are not ruled out. Suppose both b_2 and b_3 write to x and have an edge to box b_1 that reads x. In this case, if x is write-shared, the SVID cannot be consistent. If x is write-exclusive, then both b_2 and b_3 must belong to

the same process, and thus, are linearly ordered according to the visual vertical order. If b_2 appears before b_3 in this order, then the edge from b_2 to b_1 is redundant.

If there are no write-shared variables, i.e., if the SVIDs use the one-to-many communication paradigm, then consistency can be verified in linear time.

Theorem 1 (Consistency check: write-exclusive case) *If M is an SVID such that all variables are write-exclusive, then the problem of checking consistency of M can be solved in linear time.*

Proof: If all variables of M are write-exclusive, then M is consistent iff the relation $(< \cup E)^*$ is acyclic. Thus, checking inconsistency reduces to finding a cycle in a graph, and can be performed by standard linear-time search algorithms. □

One-to-many communication is the typical case for (asynchronous) hardware applications where connecting the outputs of two gates is not allowed. An example of an inconsistent SVID for this paradigm is shown in Figure 2 (there is an implied cycle between b_{13} and b_{23}).

If there are also write-shared variables, i.e., if the SVIDs also use the many to one communication paradigm, then checking consistency can be difficult. If x is a write-shared variable written in vertex u and read in vertex v, then the consistency requirement says that any other vertex w that writes to x should come either before u or after v. That is, we want to add an edge from w to u or from v to w. If only one of these edges is consistent with the partial-order $(< \cup E)^*$ then we can resolve the dependence efficiently. This is likely to be the typical case. For instance, in sample scenario M_2, the consistency requirement says that the box b_{12} should either precede b_{22} or follow b_{13}. The latter is inconsistent with the vertical order. Hence, we can resolve the disjunctive dependency conclusively by adding an edge from b_{12} to b_{22}. If both the edges are consistent, we can make a greedy choice, but backtracking may be necessary, leading to exponential worst-case complexity

Theorem 2 (Consistency check: general case) *Checking the consistency of an SVID with write shared variables is NP-complete.*

Proof: Membership in NP is obvious since checking whether a guessed linearization satisfies all the requirements of consistency is easy. For hardness, the proof is by reduction from the problem of checking sequential consistency. Consider the case when all variables are shared, and all boxes involve a single read or a single write. Checking consistency, then, corresponds to verifying whether the local views of individual processes are sequentially consistent. Since the edges match reads with writes, this is a restricted case of sequential consistency, which is known to be NP-hard [8]. □

Henceforth we will assume that we are dealing only with consistent SVIDs. Note that an SVID has multiple linearizations, and thus, provides a more succinct representation than linear traces.

3. Refinement

Given an SVID S as a high-level requirement, one may want to add more details to the requirement leading to another SVID I. In this context, given an implementation SVID I and a specification SVID S, it is important to know if I *refines (or implements)* S, written as $I \leq S$, in a mathematically precise way. We are mainly interested in a refinement notion that has a concrete, syntactic counterpart that would guide the users to add details in a visual, yet formal, way. Intuitively, there are three simple ways to refine an SVID: (1) by moving arrows and adding empty boxes, (2) by splitting the vertices and (3) by splitting the processes [2].

Moving arrows and adding empty boxes. The write exclusive variables of a process p that are updated in a box b_1 maintain their value down the process line as long as they are not updated in another box b_2. This is not only an expected behavior for SVIDs but it also allows to move the arrows exporting only write exclusive variables in order to simplify the SVID, by minimizing the number of arrows crossing each other. The transformation of moving arrows can restrict the set of allowed linearizations, and leads to a refined SVID. Moreover, it allows to introduce empty boxes if this further simplifies the SVID. In the SVID M_2^1 in Figure 2, we show how to move the arrow exporting the flag f_1 of the SVID M_2 of Figure 2.

Splitting vertices. One may split a box that contains a set of reads and writes in several successive boxes placed on the same process line and such that each contains a disjoint subset of the original set of reads and writes. As an example of this transformation, the SVID M_2^2 in Figure 2 refines the SVID M_2^1. Splitting boxes may be applied repeatedly until all boxes contain either only one read or only one write operation for a single variable.

Splitting processes. The third type of refinement we consider is splitting processes into subprocesses such that the set of variables of each subprocess is contained in one of the original processes (thus, implementation processes cannot aggregate distinct variables of different processes). During the splitting, a write shared variable may become write exclusive. However, a write exclusive variable cannot become write shared. Similarly to splitting vertices, splitting processes may be applied repeatedly until each process writes only one variable, distinct from the variable associ-

[2] In the full paper we define a more general refinement that allows to augment an SVID with additional variables, boxes, and/or arrows.

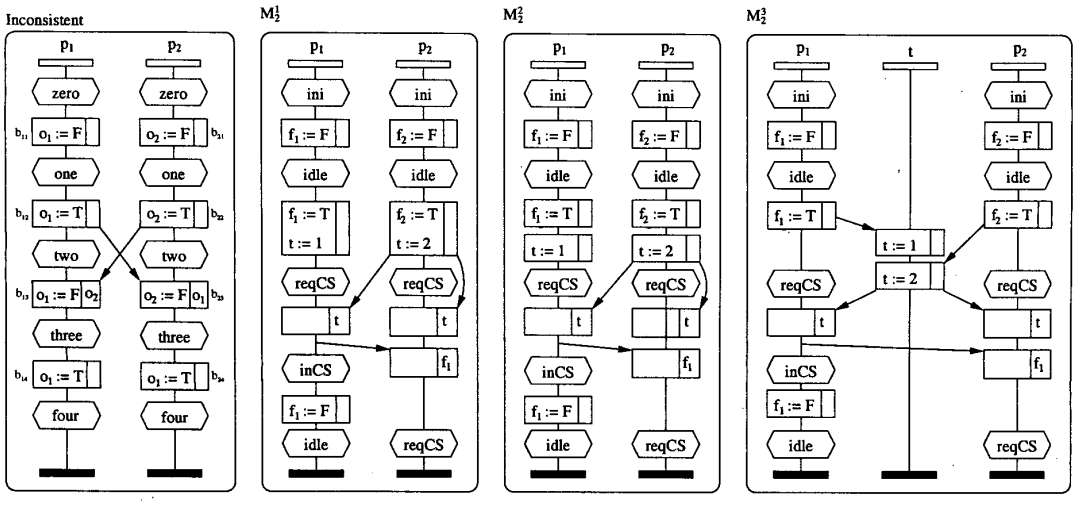

Figure 2. Inconsistency and refinement by moving arrows and splitting vertices and processes

ated to the other processes. An example of this kind of refinement is the SVID M_2^3 in Figure 2 obtained by refining M_2^2. In this SVID, the write shared variable t has been factored out both from p_1 and p_2 and transformed into a write exclusive variable. This SVID mimics the way Peterson's protocol would be implemented with MSCs. In this case, a shared resource like t is written by sending it messages (mimicked with SVIDs by a write followed by an event arrow) and it is read by getting from it a message (mimicked with SVIDs by a read arrow). Since this SVID is a refinement of the original SVID, it should be clear that SVIDs offer in general a very high level of abstraction.

Now let us proceed to formally define the notion of refinement.

Definition 3 (Refinement) *A shared variables interaction diagram* I refines *a shared variables interaction diagram* S, *written* $I \leq S$ *if:*

Variables. *The set of variables in* S *and* I *is the same:* $S.X = I.X$.

Processes. *For every implementation process* $p \in I.P$ *and every specification process* $q \in S.P$, *either* $p.X \cap q.X = \emptyset$ *or* $p.X \subseteq q.X$. *In the latter case, write shared variables in* S *may become write exclusive in* I, *but write exclusive variables in* S *cannot become write shared:* $p.X_s \subseteq q.X_s$ *and* $p.X_e \subseteq q.X_w$.

Boxes. *There is a* surjective map r *from the vertices of* I *to the vertices of* S *such that for each vertex* v *of* S, $v.X_w = \cup_{r(w)=v} w.X_w$, $v.s = \cup_{r(w)=v} w.s$, *and* $v.X_r = \cup_{r(w)=v} w.X_r$ [3].

[3] In the more general notion of refinement, all equalities can be made subsets, allowing the implementation to add more variables.

Dependency. *The specification partial order* $(S.E \cup S.<)$ *is included in the image* $r(I.E \cup I.<)$ *of the implementation partial order under the map* r. *That is, whenever a specification vertex* u *is related to* v *according to* $(S.E \cup S.<)$, *there exist implementation vertices* u' *and* v' *such that* $r(u') = u$, $r(v') = v$, *and* u' *is related to* v' *according to* $(I.E \cup I.<)$. □

By this definition, we have

$$M_2^3 \leq M_2^2 \leq M_2^1 \leq M_2.$$

For example, let the boxes in M_2^1 be denoted by $m_{11}, m_{12}, m_{13}, m_{14}, m_{21}, m_{22}, m_{23}, m_{24}$ and the boxes in M_2^3 be denoted by $n_{11}, n_{12}, n_{13}, n_{14}, n_{21}, n_{22}, n_{31}, n_{32}, n_{33}, n_{34}$, as shown in Figure 3. Define the implementation map r such that it maps n_{11} to m_{11}, n_{31} to m_{21}, n_{12} and n_{21} to m_{12}, n_{32} and n_{22} to m_{22}, n_{13} to m_{13}, n_{33} to m_{23} and finally n_{14} to m_{14} and n_{34} to m_{24}. Then it is easy to see that the dependency condition is satisfied and that the process condition is satisfied too. Hence, M_2^3 refines M_2^1.

Note that the vertical dependency (n_{21}, n_{22}) in M_2^3 becomes an arrow (m_{12}, m_{22}) in the image $r(M_2^3)$. Similarly, the arrows (n_{12}, n_{21}) and (n_{32}, n_{22}) in M_2^3 become self-loops in $r(M_2^3)$. All these arrows are new when compared with M_2^1.

Given an "implementation" SVID I and a "specification" SVID S, we would like to determine algorithmically if $I \leq S$ holds. According to the definition of refinement, we first need to verify the requirements concerning variables and processes, both of which are stratighforward to check.

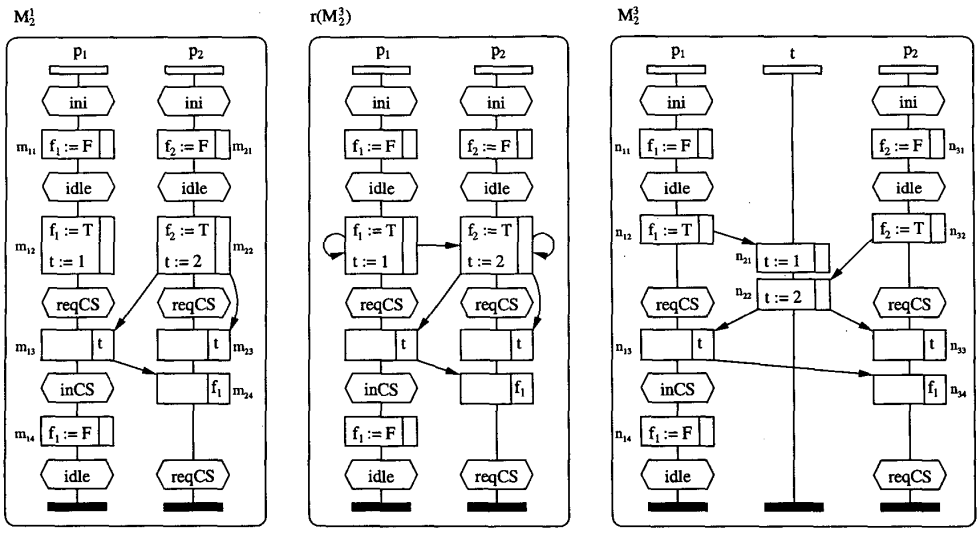

Figure 3. Refinement image by a mapping r

Then, the the algorithm has to guess (or synthesize) a vertex map r and to check the dependencies. This, in general, can be computationally difficult if the actions appearing in an implementation vertex can appear in multiple processes.

Theorem 3 (Checking refinement: general case) *Given two SVIDs S and I, deciding whether $I \leq S$ holds is NP-complete.*

Proof: In the general case, membership in NP is obvious since once the refinement map r is guessed, checking whether all dependencies are preserved can be done polynomial-time. For NP-hardness, it is known that given an MSC whose vertices are labeled with symbols in an alphabet Σ, and a string w over Σ, the problem of determining whether w is a possible linearization of vertices of the MSC is NP-complete [4]. This problem can be reduced to refinement checking problem, where I contains a single process. □

If every specification variable is written by at most one process and read by at most one process (i.e. the communication is point-to-point) then each action in the implementation can be mapped to vertices of a uniquely determined process. Since vertices of a single process are totally ordered, there is no ambiguity in determining the refinement map. Once the refinement map is determined, the requirement about dependencies can be checked by computing the transitive closure of the imeplementation partial-order in cubic time.

Theorem 4 (Checking refinement: exclusive case) *Given two SVIDs S and I such that each variable is read by a single process and written by a single process, the refinement relation $I \leq S$ can be determined in $O(n^3)$, where n is the number of vertices in I.*

In the standard notion of refinement for processes, implementation is obtained from specification by adding more details, and typically, by restricting the set of observable behaviors. The notion of implementation for SVIDs allows splitting processes and adding constraints, and thus, constraining the set of valid linearizations. However, it also allows splitting of vertices, thus changing the notion of atomicity and "increasing" the possible interleavings. While this may seem counter to the traditional notion of hierarchical development of systems, it seems intuitive for developing requirements in a systematic manner by refining the atomicity assumptions[4].

4. Comparison with Other Formalisms

As we discussed in the introduction, the purpose of shared variables interaction diagrams (SVIDs) is to describe exemplary interactions (scenarios) of concurrent systems that communicate via shared variables. Hence, they play the same role for these systems as message sequence charts (MSCs) or interaction diagrams (in UML) play for concurrent systems that communicate via message passing. While scenarios involving shared variables appear in many con-

[4]One can reconcile these two notions of implementation by observing that, under the refinement map the implementation has more constraints and therefore less linearizations than the specification.

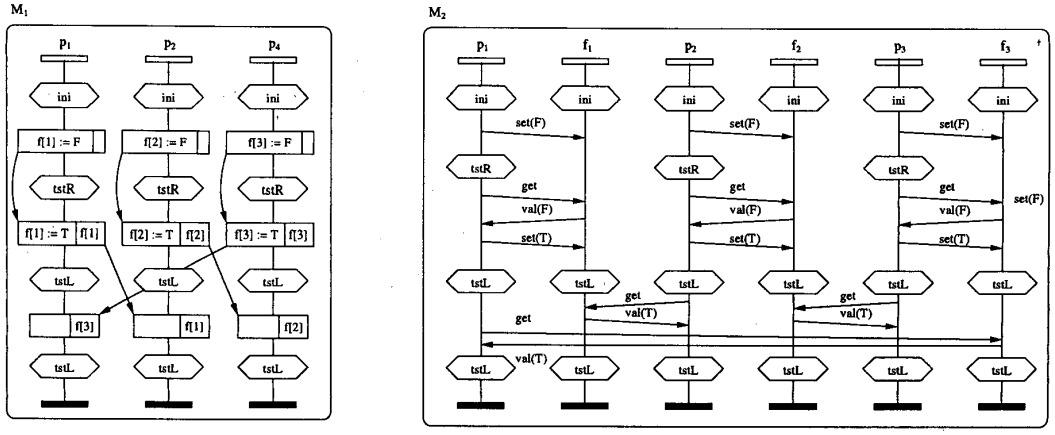

Figure 4. Deadlock for three dining philosophers

texts, there seems to be no effort to formally define a notation for our intended purposes. One exception is [9], in which the authors define a notation for describing scenarios for hybrid systems in which synchronous communication can be captured by global conditions over variables.

Since MSCs are already standardized by the ITU consortium and very popular (e.g. in the specification of telecommunication protocols) one might wonder if MSCs could not be used instead of SVIDs for shared variables systems, too. Indeed, by associating with each global variable a separate process (vertical line) and introducing three messages `set(x)`, `get` and `val(v)` one can model reading and writing these variables. However, this is an indirect way of dealing with the read and write primitives and the more shared variables the system contains, the more complex the MSCs become.

To illustrate this point, let us consider the dining philosophers example (see, for instance, [15]). There are n philosophers seated around a table, usually thinking. Between each pair of philosophers is a single fork. From time to time, any philosopher might become hungry and attempt to eat. In order to eat, the philosopher needs exclusive use of the two adjacent forks. After eating, the philosopher relinquishes the two forks and resumes thinking.

In a wrong symmetric solution, each philosopher will wait first for the right fork and after getting it will wait for the left fork. However, if all philosophers pick the right fork first, they will wait forever for the left fork. This situation is illustrated for three philosophers in Figure 4 where the left hand side is the scenario presented as an SVID and the right hand side is the scenario presented as an MSC. The MSC solution has more processes, more arrows and more crossings. We believe, this makes it harder to read and understand. Trying to model message passing systems in an indirect way with SVIDs would probably lead to more complex solutions too.

From a visual point of view (as boxes connected by arrows), SVIDs are also related to acyclic versions of Petri nets and data flow diagrams (DFDs). Unlike these formalisms, an SVID denotes a single partially-ordered execution, and has an implicit notion of shared state.

5. Conclusions

We have presented a formal notation for visual description of scenarios in distributed systems when the communication is via shared variables. We have also presented a natural and precise way of relating such scenarios via the notion of refinement. SVIDs can be useful for writing design requirements. Many model checkers, such as SPIN [10] and MOCHA [1], support shared-variables communication and show the counter-examples in a graphical format like SVIDs. Consequently, SVIDs can also be used to compare requirements with executions of models.

SVIDs are also a promising specification and debugging formalism for concurrent systems written in Java. This is one of the few languages that allows process creation and process control. In Java processes are instances of a special class called `Thread` and run in an asynchronous way. Threads interact by using the shared memory paradigm.

To obtain richer specifications using SVIDs, we will need to consider *high-level* SVIDs in the spirit of high-level MSCs (a high-level MSC is basically a finite graph whose nodes are labeled with basic MSCs). To be able give semantics to such specifications, we need to define concatenation of SVIDs that accounts for the shared state.

References

[1] R. Alur, L. de Alfaro, R. Grosu, T.A. Henzinger, M. Kang, R. Majumdar, F. Mang, C.M. Kirsch, and B.Y. Wang. MOCHA: A model checking tool that exploits design structure. In *Proceedings of 23rd International Conference on Software Engineering*, 2001.

[2] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proceedings of 22nd International Conference on Software Engineering*, pages 304–313, 2000.

[3] R. Alur, G.J. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.

[4] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *CONCUR'99: Concurrency Theory, Tenth International Conference*, LNCS 1664, pages 114–129. Springer-Verlag, 1999.

[5] H. Ben-Abdallah and S. Leue. MESA: Support for scenario-based design of concurrent systems. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1384, pages 118–135, 1998.

[6] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1997.

[7] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Proceedings of the Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, 1997.

[8] P. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[9] R. Grosu, I. Krueger, and T. Stauner. Hybrid sequence charts. In *ISORC'2K, the 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 104–111, 2000.

[10] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[11] G.J. Holzmann, D.A. Peled, and M.H. Redberg. Design tools for for requirements engineering. *Lucent Bell Labs Technical Journal*, 2(1):86–95, 1997.

[12] I. Krueger, R. Grosu, P. Scholz, M. Broy, From MSCs to Statecharts. In *Distributed and Parallel Embedded Systems*, Kluwer Academic Publishers, 1999.

[13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[14] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley, 2000.

[15] N. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[16] A. Muscholl, D. Peled, and Z. Su. Deciding properties of message sequence charts. In *Foundations of Software Science and Computation Structures*, 1998.

[17] J. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[18] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw-Hill, 1999.

[19] E. Rudolph, P. Graubmann, and J. Gabowski. Tutorial on message sequence charts. In *Computer Networks and ISDN Systems – SDL and MSC*, volume 28. 1996.

[20] R.J. van Glabbeek and U. Goltz. Refinement of Actions in Causality Based Models. In *Comparitive Concurrency Semantics and Refinement of Actions*, pages 161–203, 1989.