Block-based Atomicity in Message-passing Distributed Programs

Eslam Al Maghayreh

Computer Science Department
Faculty of Information Technology and Computer Science
Yarmouk University
Irbid 21163, Jordan.
eslam@yu.edu.jo

Abstract

In this paper, we have studied the notion of atomicity in message-passing distributed programs. The difference between the notion of atomicity in shared memory programs and message-passing programs has been discussed in detail. Due to the nature of communication in message-passing programs, the atomicity requirement in shared memory programs (serializability) does not guarantee the same level of atomicity in message-passing programs. We have referred to the atomicity level guaranteed by the serializability requirement in message-passing programs as the weak atomicity. We have defined the requirements that guarantee a level of atomicity in message-passing programs similar to that in shared memory programs. We have referred to it as the strong atomicity requirement. Both of weak and strong atomicity have been formally defined.

The notion of atomicity can be employed to significantly reduce the state space to be considered in verification. Moreover, atomicity violations in a run typically indicate the presence of program bugs. Consequently, the concepts presented in this paper can be exploited to reduce the cost of the verification of message-passing distributed programs.

Keywords: Distributed programs, Atomicity, Verification, State space reduction, Testing, Debugging.

1. Introduction

Distributed programs involve a set of processes that communicate with each other through passing messages. A major challenge is to simplify the verification of distributed programs. The notion of atomicity can be exploited in modeling distributed programs as programs that execute a set of atomic actions. This model can significantly simplify the verification of these programs [1, 2, 3, 4].

Atomicity is a well known correctness requirement for multi-threaded programs. It requires that concurrent invocations of a set of methods be equivalent to performing the invocations in some sequential order [5, 6, 7, 8, 9, 10, 11]. This is similar to serializability in transaction processing. We will briefly highlight some of the related work in this sub-area. However,
in this paper we will concentrate on studying the atomicity of actions in message-passing distributed programs.

Unexpected interactions between processes may result in incorrect overall behavior of message-passing distributed programs. Some researchers tried to model incorrect interactions between processes by defining message races and developing algorithms for detecting and reporting them [12, 13, 14]. However, message races are not always harmful, and hence they are not enough to identify all kinds of unexpected interactions between processes. For example, in a master-slave program it is normally expected to have message races that are not harmful. Consequently, a stronger property is actually needed to capture these unexpected interactions. The notion of atomicity can be employed to develop more efficient algorithms to detect unexpected interactions between different processes (atomicity violations).

Errors in the communication part of a message-passing program may result in violating the atomicity requirements of the computational part of a method (or code block in general). Modeling the atomicity of these code blocks and developing efficient algorithms to detect atomicity violations will help programmers in locating and fixing communication bugs in their programs. Moreover, the notion of atomicity can be significantly exploited in simplifying the verification of distributed programs. In verification, we have to consider all the possible interleavings of a distributed program to verify that certain properties are satisfied. If we can guarantee that some code blocks are atomic, then it will be enough to consider only the interleavings where the events belonging to each atomic code block appear in some sequential order without interleaving with the events of other process [1, 15].

The rest of the paper is organized as follows: Section 2 briefly presents some of the related works in this area. Section 3 presents a formal model of a run of a message-passing distributed program. The notions of weak and strong atomicity are informally presented in Section 4. Section 5 presents a formal model of both weak and strong atomicity in message-passing programs. Section 6 describes an algorithm to detect atomicity violations of both types. Finally, a summary of the main conclusions and possible future works are presented in Section 7.

2. Related Works

Atomicity is a common correctness requirement for multi-threaded programs. Several researchers have defined and modeled the atomicity of a code block in multi-threaded programs [9, 10, 11]. It requires that concurrent invocations of a set of methods be equivalent to executing the invocations in some sequential order.

Verifying the correctness of multi-threaded programs is a very hard task due to the potential for unexpected and nondeterministic interactions between threads. Some researchers have proposed techniques for detecting race conditions, a situation where two threads simultaneously access the same data variable, and at least one of the accesses is a write [16, 17]. However, the absence of race conditions does not guarantee the absence of errors due to unexpected thread interactions. Consequently, we need a property to verify the correctness of a multi-threaded program at higher level of granularity, namely, the atomicity of code blocks [11]. If a code block is atomic, we can safely verify the correctness of a program at a higher level of granularity, where each atomic code block is executed in a single step [11].

A code block can be considered atomic if, for every execution of the program, there is an equivalent execution that have the same overall behavior, where the statements of the atomic
code block are executed in sequential order without interleaving with the execution of any other code block [9, 10, 11, 18, 19].

In [2], Lamport has introduced a theorem on atomicity to simplify verification of distributed and parallel systems. Based on this theorem, we can group a sequence of statements in a distributed program together and treat them as an atomic action under some stated conditions. According to the conditions stated in the theorem, an atomic action can receive information from other processes, followed by at most one externally visible event (for example, modifying a variable relevant to some of the properties to be verified) before sending information to other processes. Based on this theorem, we can abstract a distributed program and hence the cost of verifying it can also be reduced.

In [3, 18, 19], the authors have formally defined the notion of atomic actions in message-passing distributed programs. They have exploited the atomicity concept in reducing the state space to be considered in runtime verification of message-passing distributed programs. In this paper, we distinguish between the notions of atomicity in shared memory distributed programs and in message-passing distributed programs. We will show that due to the communication nature in message-passing programs, there will be two levels of atomicity for an action (weak and strong). The two levels of atomicity will be formally defined and an algorithm to detect atomicity violations will be described.

The following section describes a formal model of a run of a message-passing distributed program.

3. A Formal Model of a Run of a Message-passing Distributed Program

We assume a message-passing distributed program consisting of \( n \) processes denoted by \( P_1, P_2, ..., P_n \) and a set of unidirectional channels. The execution of a statement in the distributed program is called an event. An event can be an internal computational event or an external message event (send/receive). Events are related by either their execution order in a process or message send/receive relations across processes. The happened-before relation \( (\rightarrow) \) between events applies to all events executed [20]. The data dependency relation \( D \rightarrow \) is defined between the events that belong to the same process only. \( e_i D \rightarrow e_j \) if \( e_j \) is data dependent on \( e_i \).

**Definition 1** A run of a distributed program is an event structure \( \langle E, \rightarrow, D \rightarrow \rangle \), where \( E \) is the set of events executed, \( \rightarrow \) and \( D \rightarrow \) are the happened-before relation and the data dependency relation among the events in \( E \).

A run of a message-passing distributed program can be depicted in terms of a two dimensional space-time diagram. Space is represented in the vertical direction and time in the horizontal direction. Circles are used to represent events in the space-time diagram. The transmission of a message is represented by a directed edge going from the send event to the corresponding receive event. The space-time diagram shown in Figure 1 represents a run of a message-passing distributed program involving two processes.

In a given run of a message-passing distributed program, \( a \rightarrow b \) if and only if there is a directed path in the corresponding space-time diagram from event \( a \) to event \( b \). The happened-before relation is a partial order relation on the set of events executed. Consequently, two
events $a$ and $b$ may not be related by the happened-before relation, in this case we say that $a$ and $b$ are concurrent (denoted by $a \parallel b$). For example, in Figure 1, $e_1 \parallel e_7$ because $\neg(e_1 \rightarrow e_7)$ and $\neg(e_7 \rightarrow e_1)$.

Definition 2  A consistent cut $C$ of a run $\langle E, \rightarrow, D \rightarrow \rangle$ is a finite subset $C \subseteq E$ such that if $e \in C$ and $e' \rightarrow e$ then $e' \in C$.

Each consistent cut $C$ is associated with a global state of the program represented by the values of the program variables and channel states attained upon the completion of the execution of the events in $C$. According to [21], the set of global states of a given run endowed with set union and set intersection operations forms a distributive lattice, referred to as the state lattice. Using the state lattice, we can verify whether a run of a given distributed program satisfies the necessary properties or not (runtime verification).

Figure 2 depicts the state lattice $L$ associated with the run depicted in Figure 1. Each global state can be labeled by the most recent event executed in each process upon reaching it. For example, $(e_2, e_{11})$ is the state reached after executing event $e_2$ in $P_1$ and event $e_{11}$ in $P_2$.

4. Block-based Atomicity in Message-passing Distributed Programs

According to [18, 19], a code block can be labeled as an atomic code block based on design time knowledge. Each execution of a labeled code block is called an action. An action is called atomic if the events belonging to that action in any execution trace can be reordered so that they appear in sequence without interleaving with any event of any other process. A run of a given distributed program is called atomic if and only if all the actions in the run are atomic actions. In [18], the authors have presented an algorithm to check whether a run of a given distributed program is atomic or not. If a run is proved to be atomic then verifying that the given run satisfies certain properties will be much easier as it has been illustrated in [1, 4, 18, 19, 22].

In general, atomicity requires that any concurrent execution of a set of actions is equivalent to executing those actions in some sequential order without interleaving. The usual method of ensuring atomicity in the presence of concurrency is to guarantee serializability; namely, actions are scheduled in such a way such that their overall effect is as if they had been executed in some sequential order [1, 11].
Atomicity in shared memory programs means that we can view the execution of an atomic action as the execution of a single event. Consequently, if a variable has been modified more than once inside an atomic action, then only the final version is made visible to other actions. However, in message-passing distributed programs, the intermediate values of a given variable may be visible to other actions even if the two actions satisfy the above mentioned requirement of atomicity (serializability). This is due to the fact that message-passing communication incur message delay and the messages may reside in the sender and/or receiver buffers for sometime. Figure 3 illustrates this fact (Rectangles are used to depict actions).

Figure 3 (a) depicts the space-time diagram corresponding to a run of a distributed program involving two processes. Each process executes a single action. The two actions (A and B) in Figure 3 (a) overlap in time. However, Figure 3 (b) shows that the two actions satisfy the serializability requirements and hence the events in the two actions can be reordered such that all of the events in action A complete before any event in B starts. However, serializability here is not equivalent to serializability in shared memory programs. If we assume that P1 and P2 are two threads and x is a shared variable between them then according to Figure 3 (b) action B will only see the final value of x written by action A. But in message-passing we can see that action B sees the two values of x (0 and 1) even though it has started after A has completed the execution of all of its events. This is due to the fact that the communication between the two actions is through passing messages not through shared memory.

There is a clear separation between communication and computation in message-passing programs. The communication part involves a set of statements to send/receive values to/from other processes. An action that involves only computational events is guaranteed to be atomic due to the fact that message-passing programs do not share variables, and hence the interaction between processes occurs in the communication part. The atomicity of an action could be
violated when the processes interact with each other in an unexpected manner.

Serializability is enough to ensure that an action is atomic in shared memory programs; this is due to the nature of communication in shared memory programs where computation is not separated from communication. An output by an action (altering a shared variable) is immediately visible to all other processes, while in message-passing programmes an output of an action may reside in buffers for some time.

Based on the above scenario we can see that if an action in a message-passing program wants to have the same level of atomicity as actions in shared memory programs, then it must satisfy the following requirements in addition to serializability (serializability alone is not enough).

1. If the value of a variable is sent out, then this variable should not be updated and sent out again within the same action. This condition guarantees that intermediate values of an atomic action are not visible to other actions. Action $A$ in Figure 3 violates this requirement.

2. Input values required for an action should be ready before any computation statement is executed. An atomic action should have the ability to be executed as if it is a single event without any need to wait for any thing. If the computational events in an action start before all the inputs are ready, the action may have to stop at some point waiting for the rest of the input values to be received. Action $B$ in Figure 3 violates this requirement.

We will refer to these requirements as the **strong atomicity requirements**. Based on the above requirements, it is not enough to show that the events of an action in a message-passing
program are serializable to consider it atomic. A violation of the strong atomicity requirement occurs if an action that is supposed to be atomic, does not satisfy any of the above mentioned requirements. Figure 3 is an example of strong atomicity violation. More details about strong atomicity will be given later.

However, it might not be necessary for all actions to satisfy these requirements (in some actions in message-passing programs, it might be fine for other actions to view the intermediate values of some variables). Design time knowledge can help us in figuring out whether an action has to satisfy these requirements or not. If an action does not need to satisfy the above requirements then it has to satisfy the serializability requirement only in order to consider it atomic. We will refer to the serializability requirement in message-passing programs as the weak atomicity requirement. Serializability in shared memory is equivalent to strong atomicity in message-passing programs. In our work we will consider message-passing programs. If an action satisfies the strong atomicity requirements, then by default it satisfies the weak atomicity requirement but not vice versa. Subsequent discussions will give more clarifications.

The authors in [18] considered weak atomicity and developed an online algorithm to detect its violation in a given run of a distributed message-passing program. In our work we will distinguish between weak and strong atomicity and we will describe an algorithm to report atomicity violations of both types. In the following section, we will formally define the concepts presented in this section.

5. A Formal Model of Weak and Strong Atomicity

In this section, we will introduce a formal definition of weak and strong atomicity requirements that have been introduced informally in the previous section.

According to [18], a code block can be labeled as an atomic code block based on design time knowledge. We will assume that a distributed program consists of a set of code blocks. An execution of a code block is called an action.

Figure 4 depicts the actions of the run shown in Figure 1. The events of the run belong to five actions, \(A, B, C, D \) and \(E\). For example, the events of process \(P_1\) belong to three actions \((A, B \) and \(C\)). Each action has an input signature and an output signature.

Let \(R\) represents the set of receive events in an action.

Let \(S\) represents the set of send events in an action.

Let \(C_{in}\) represents the set of computation events in an action.

Let \(V\) represents the set of variables used in an action.

![Figure 4. The actions of the run shown in Figure 1.](image)
**Definition 3** An input signature $I_s$ of an action $A$ is the set of variables updated during the execution of $A$ using values received from other processes. $I_s$ will be a set of pairs: the variable and the corresponding receive event.

$$I_s = \{(v_1, r_1), (v_2, r_2), \ldots\}$$ such that for every pair $(v_x, r_x)$, the value received in $r_x \in R$ is used to update $v_x \in V$.

**Definition 4** An output signature $O_s$ of an action $A$ is the set of variables whose values have been sent to other processes during the execution of $A$. $O_s$ will be a set of pairs: the variable and the corresponding send event.

$$O_s = \{(v_1, s_1), (v_2, s_2), \ldots\}$$ such that for every pair $(v_x, s_x)$, the value of variable $v_x \in V$ is sent to another process by the send event $s_x \in S$.

**Definition 5** An action satisfies the **Weak Atomicity** requirements if the events that belong to it in a given execution trace can be reordered such that they appear in some sequential order without interleaving with any event of any other process.

**Definition 6** An action $A$ satisfies the **Strong Atomicity** requirements if it satisfies the weak atomicity requirements in Definition 5 and the events that belong to it in a given execution trace can be reordered such that they have the form described by the following regular expression

$$A = R^* C_m^* S^*$$

That is the set of all receive events appears first followed by the set of all computation events and finally the set of all send events appears last.

Weak atomicity requires only serializeability. In shared memory programs weak atomicity is equivalent to strong atomicity because serializeability guarantees that intermediate values are not visible to other actions. In message-passing programs an action may have to satisfy the strong or just the weak atomicity requirements; this can be extracted from design time knowledge.

The following are some scenarios of strong atomicity violation:

- A variable $x$ has been sent two times and in between variable $x$ has been modified. Consequently, two different versions of the same output variable are visible to other processes (not only the final version). As a result, the events of the action cannot be reordered according to the formula shown in Definition 6. Action $A$ in Figure 3 depicts an example of this strong atomicity violation scenario. This scenario can be defined formally as follows:

  - $\exists ((x, s_1) \in O_s \land (x, s_2) \in O_s)$ and variable $x$ has been updated by some computational event between $s_1$ and $s_2$.

- A variable $x$ has been updated two times using values received from other processes and in between variable $x$ has been used in some computational event. Consequently, two different versions of the same variable have been used by the computational part. As a result, the events of the action cannot be reordered according to the formula shown in Definition 6. Action $B$ in Figure 3 depicts an example of this strong atomicity violation scenario. This scenario can be defined formally as follows:
Figure 5. The atomic state lattice associated with the run shown in Figure 4.

\[ \exists ((x, r_1) \in I_s \land (x, r_2) \in I_s) \] and variable $x$ has been used by some computational event between $r_1$ and $r_2$.

The work in [18] considers weak atomicity only. An action is called atomic if the events belonging to that action in any execution trace can be reordered so that they appear in sequence without interleaving with any event of any other process. A run of a given distributed program is called atomic if and only if all the actions in the run are atomic action [18].

The authors of [18] proved that if a run is atomic then the properties of the run can be verified on an atomic state lattice rather than the much more complex state lattice corresponding to all of the events in a run. The atomic state lattice associated with the run shown in Figure 4 is shown in Figure 5. It is clear that this lattice is much smaller than lattice $L$ (shown in Figure 2) corresponding to all of the events in the run. Consequently, the notion of atomic actions can significantly reduce the cost of verification.

With atomic actions as the basis in viewing a run, checking the satisfaction of a property in a run of a given distributed program (runtime verification) involves two steps. First, the atomicity of the run needs to be checked. If the run is not atomic, then there is interference among the actions associated with the program and debugging should proceed with analyzing the source of this error. This error could arise due to inappropriate synchronization among the processes. Second, if the run is atomic, then the application specific properties of the program can be checked based on the atomic state lattice.

In the following section, we will describe an algorithm to detect the violations of both weak and strong atomicity requirements in a run of a given distributed program.

6. Detecting Atomicity Violation

In this section we will describe an algorithm to detect atomicity violations. Based on the definitions of weak and strong atomicity (Definitions 5 and 6), the detection algorithm has to perform the following two abstract steps:

1. Ensuring that the events of the given run of a distributed program can be reordered based on the happen-before relation such that the events that belong to each atomic action appear in sequence without interleaving with any event of any other process. If
the events of any atomic action cannot be reordered to satisfy this condition, then a violation of the weak atomicity requirement should be reported.

2. Reordering the events belonging to the same action based on the data dependency relation \( (D \rightarrow) \) so that the events of each action has the form described by the regular expression shown in Definition 6. If the events of any action (that is supposed to satisfy the strong atomicity requirement) cannot be reordered to satisfy this condition, then a violation of the strong atomicity requirement should be reported.

In [18], the authors have presented an efficient dynamic algorithm to check whether a run of a given distributed program satisfies the weak atomicity requirement or not. The space-time diagram of any run of a given distributed program is actually an acyclic directed graph where each node in this graph represents an event and the edges in this graph represent the happened-before relation among the events. The algorithm presented in [18] exploits this fact and abstracts the original space-time diagram of the run under consideration such that the nodes in the abstracted space-time diagram represent actions instead of events. If the abstracted space-time diagram is also an acyclic directed graph then we can conclude that all the actions in the run satisfy the weak atomicity requirement. The details of the weak atomicity violation detection algorithm along with its proof of correctness can be found in [18].

Figure 6 depicts the abstracted space-time diagram of the space-time diagram shown in Figure 4. It is clear that the abstracted space-time diagram does not have any cycle. Consequently, we can conclude that the run shown in Figure 4 satisfies the weak atomicity requirement.

Each cycle in the abstracted space-time diagram (if any) indicates that an action has violated the weak atomicity requirement. However, if the abstracted space-time diagram is acyclic, then we can move to the second step and reorder the events in each action (that must satisfy the strong atomicity requirement) based on the data dependency relation. After reordering, if all the actions have the form described in Definition 6 then we can say that all the actions in the given run satisfy the expected atomicity requirements.

In our example depicted in Figure 4, actions \( A, B, C \) and \( D \) satisfy the strong atomicity requirements without the need to perform any reordering since the events of each of these actions conforms with the regular expression in Definition 6. However, the events of action \( E \) does not conform with the required regular expression, but we can reorder its events to match the required regular expression. We can move the first receive event in \( E \) to appear before the first computational event since there is no data dependency between the two events. Conse-
quentlly, the resulting sequence of events conforms with the required regular expression after reordering, and hence we can conclude that action $E$ satisfies the strong atomicity requirement.

The presence of atomicity violations is a strong indication of the existence of communication errors in the distributed program under consideration. The programmer has to check the code blocks belonging to each of the actions that violates the atomicity requirements and fix the bugs. If there are no atomicity violations, then we can be sure that the abstracted space-time diagram is a legal abstraction of the original space-time diagram. Consequently, any other property that we may want to verify can be verified based on the abstracted space-time diagram rather than the much larger original space-time diagram.

7. Conclusions and Future Work

In this paper, we have formally defined the notions of weak and strong atomicity in message-passing distributed programs. An algorithm to detect atomicity violations has been described. Atomicity violations are considered as a strong indication of the existence of software bugs. Moreover, the notion of atomicity can be exploited to significantly reduce the state space to be considered in verification. Consequently, the concepts presented in this paper can be exploited in developing more dependable distributed programs.

Two approaches can be used to detect atomicity violations, the off-line approach and the online approach. The off-line approach collects the necessary information in a trace file during the execution and later analyzes it to detect atomicity violations. The online approach detects and reports atomicity violations during the execution of the program. The online approach has the advantage of early violation reporting and avoiding the need for large trace files. However, it has the disadvantage of incurring more space and time overhead during execution. The off-line approach does not incur significant space and time overhead during execution, but it has the disadvantage of generating very large trace files. The work presented in this paper can be extended by comparing between the two approaches in detecting atomicity violations.

References

Author

Eslam Al Maghayreh is an assistant professor in the department of computer science at Yarmouk University (Jordan). He has received his PhD degree in Computer Science from Concordia University (Canada) in 2008, his M.Sc. degree in Computer Science from Yarmouk University in 2003, and his B.Sc. degree in Computer Science from Yarmouk University in 2001. His research interests include: Distributed systems, multi-agent systems, and runtime verification of distributed programs.