An Efficient Validation Approach for Quasi-Synchronous Checkpointing oriented to Distributed Diagnosability

Houda Khlif, Hatem Hadj Kacem, Saúl Eduardo Pomares Hernández, Ahmed Hadj Kacem, Cédric Eichler, Alberto Calixto Simón

An Efficient Validation Approach for Quasi-Synchronous Checkpointing oriented to Distributed Diagnosability

Houda Khlif[1], Hatem Hadj Kacem[1], Saúl E. Pomares Hernandez[2,3,4], Ahmed Hadj Kacem[1], Cédric Eichler[3,4] and Alberto Calixto Simón[5]

[1] ReDCAD Laboratory FSEGS, University of Sfax, Sfax, Tunisia;
houdakhlif@gmail.com; Hatem.Hadjkacem@fsegs.rnu.tn; Ahmed.Hadjkacem@fsegs.rnu.tn

[2] Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE)
Luis Enrique Erro 1, C.P. 72840, Tonantzintla, Puebla, Mexico; spomares@inaoep.mx

[3] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France; ceichler@laas.fr

[4]Univ de Toulouse, LAAS, F-31400 Toulouse, France

[5] Universidad del Papaloapan, UNPA, Av, Ferrocarril s/n, C.P. 68400,
Loma Bonita, Oaxaca, Mexico; acalixto@unpa.edu.mx

Abstract

The Autonomic Computing paradigm is oriented towards enabling complex distributed systems to manage themselves, even in faulty situations. The diagnosability analysis is a priori study through which a system can be self-aware about its current state. It is from the determination of a consistent state that a system can take some actions to repair or reconfigure itself. Nevertheless, in a distributed system it is hard to determine consistent states since we cannot observe simultaneously all the local variables of different processes. In this context, the challenge is to efficiently monitor the system execution over time to capture trace information in order to determine if the system accomplishes both functional and non-functional requirements. Quasi-Synchronous Checkpointing is a technique that collects information from which a system can establish consistent snapshots. Based on this technique, several checkpointing algorithms have been developed. According to the checkpoint properties, they are classified into: Strictly Z-Path Free (SZPF), Z-Path Free (ZPF) and Z-Cycle Free (ZCF). Checkpointing algorithms are often evaluated with regard to performance, generally through simulation. However, their correctness has been mildly studied. In this paper, we propose an efficient validation approach based on a graph transformation oriented towards the automatic detection of the aforementioned properties.

1 Introduction

As computing systems have reached a level of complexity, their management has become increasingly difficult. As a result, the initiative of autonomic computing

has been introduced to prevent human intervention and enable the system to manage itself, even in faulty situations. An Autonomic Distributed System is considered to be a set of geographically-distributed autonomic components that communicate and collaborate with each other. In general, autonomic computing has four elements: self-configuration, self-healing, self-optimization and self-protection. The diagnosability analysis is *a priori* a study through which a system can be self-aware about its current state [2]. It is from the determination of a consistent global state that a system can take some action to repair or reconfigure itself. Nevertheless, in a distributed system, it is hard to determine consistent states since we cannot observe simultaneously all the local variables of different processes [16]. In this context, the challenge involves efficiently monitoring the system execution over time in order to capture trace information. With this information, consistent global states will be determined to evaluate if the system accomplishes both functional and non-functional requirements.

Checkpointing is a well-known technique used to identify consistent global snapshots from local recorded states called checkpoints. Informally, a global snapshot is consistent if the set of checkpoints that compose it (one per process) accomplishes the following two constraints: first, all the local checkpoints in the snapshot are concurrent and no Z-path exists from one local checkpoint to another or itself [13]. This last case is called a $Z - Cycle$ (these patterns are formally defined in Section 2.2).

Intuitively, to illustrate the concurrency constraint we present the following example scenario (a Z-path example scenario is presented in Section 2.2). In order to distributively diagnose the load balancing in two servers, the servers share a common variable v. At the beginning this variable is equal to zero, which means that there is no local load, and each time its load increases or decreases, such variable is locally incremented or decremented accordingly. If a server changes its variable by more than 10 units, it will send a message to the other server to inform it of the new value. At the reception of the message, the server updates the content of its local variable v_x with the value of v piggybacked onto the message only if $v > v_x$. The safety non-functional requirement is defined by $|v_1 - v_2| \leq \delta$. If the safety requirement is satisfied, this means that the system is balanced. The problem is distributively determining at what moments in time such constraint can be consistently evaluated. In the scenario depicted in Figure 1, there are two snapshots: the first one is composed of the checkpoints C_1^2 and C_2^2, and the second one is composed of C_1^3 and C_2^3.

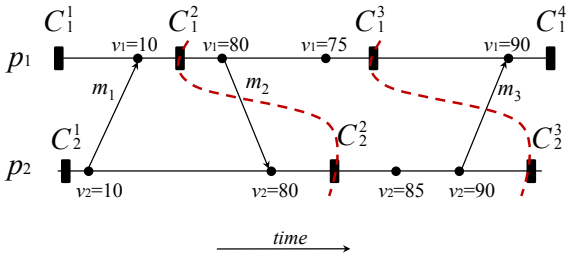


Figure 1: A Distributed Monitoring Scenario

The first snapshot is inconsistent since it does not contain all the history of the causal events at the moment of the cut (the *delivery* event of m_2 is in the cut, but not its *send* event). If we evaluate the safety constraint in this snapshot with $\delta = 50$, the result will give a false alert of unbalance since $v_1 = 80$ in C_1^2 and $v_2 = 10$ in C_2^2. The second snapshot is consistent since the entire causal history of the events is included at the moment of the cut. For this reason, only in consistent snapshots does the safety requirement need to be evaluated in order to be certain of the result.

Checkpointing algorithms are organized into three classes: asynchronous, synchronous and quasi-synchronous [12]. In asynchronous checkpointing, also known as uncoordinated checkpointing, each process takes its checkpoints independently, which leads to the domino effect. In synchronous, or coordinated checkpointing, processes coordinate their checkpoints by the addition of control messages so that a globally consistent set of checkpoints is always maintained in the system. Major disadvantages of coordinated checkpointing are: the process execution may have to be suspended during the checkpointing coordination, resulting in performance degradation, and it requires extra message overhead to synchronize the checkpointing activity. In quasi-synchronous checkpointing, or Communication Induced Checkpointing (CIC), coordination is achieved by piggybacking control information on application messages and taking forced local checkpoints in case of dangerous patterns. CIC algorithms nullify the risk of a domino effect while still allowing asynchronous execution. Several quasi-synchronous checkpointing algorithms, which propose different methods to force checkpoints and produce checkpoint and communication patterns, have been developed. They are classified into: Strictly Z-Path Free (SZPF), Z-Path Free (ZPF) and Z-Cycle Free (ZCF).

In the literature, the simulation has been the method adopted for the performance evaluation of checkpointing algorithms. Checkpointing algorithms are often evaluated with regard to performance, generally through simulation. However, few works have been designed to validate their correctness. In this paper, we propose an efficient validation approach based on graph transformation oriented towards the automatic detection of the previously mentioned properties. To achieve this, we firstly took the vector clocks resulting from an algorithm execution and modeled them into the happened-before graph (HBR graph). For efficiency sake, this graph is then reduced into the immediate dependency graph (IDR graph), i.e., the minimal causal graph. An undesirable pattern may however not always be detectable in this minimal graph. Consequently, we designed a set of transformations rules that automatically enrich this graph and proved that it can then be used to verify whether an algorithm is SZPF, ZPF, ZCF or not. An experimental study shows that, in addition to reducing the size of the input graph, using an enriched IDR graph rather than the HBR graph significantly reduces the patterns to consider by suppressing redundancies.

This paper is structured as follows. In Section 2, we define the system model and background, describe the classification of quasi-synchronous checkpointing, and define the main approaches of graph transformation. In Section 3, we present some related works. In section 4, we describe the process of our approach, and present a set of transformation rules that we have designed. Section 5 contains

a formal proof of the proposed rules and Section 6 is to discuss the performance of the obtained results. Finally, we summarize our contributions and suggest new research.

2 Preliminaries

2.1 System Model

Processes The system under consideration is composed of a set of processes $P = \{p_1, p_2, \cdots, p_n\}$. The processes present an asynchronous execution and communicate only by message passing.

Messages We consider a finite set of messages M, where each message $m \in M$ is sent considering an asynchronous reliable network that is characterized by no transmission time boundaries, no order delivery, and no loss of messages. The set of destinations of a message m is identified by $Dest(m)$.

Events We consider two types of events: internal and external events. An internal event is a unique action that occurs at a process p in a local manner and which changes only the local process state. We denote the finite set of internal events as R. The set R represents the set of relevant events. We consider only the checkpoints as internal events. We denote by C_i^x the x^{th} checkpoint of process p_i. The sequence of events occurring at p_i, between C_i^{x-1} and C_i^x ($i > 0$), is called a checkpoint interval (denoted as I_i^x). An external event is also a unique action that occurs at a process; it is seen by other processes, thus, affecting the global state of the system. The external events considered in this paper are the send and delivery events. Let m be a message. We denote by $send(m)$ the emission event and by $delivery(p, m)$ the delivery event of m to participant $p \in P$. The set of events associated to M is the set:
$E(M) = \{send(m) : m \in M\} \cup \{delivery(p, m) : m \in M \wedge p \in P\}$.
The whole set of events in the system is the finite set $E = R \cup E(M)$. The distributed computation is modeled by the partially ordered set $\hat{\mathrm{E}} = (\hat{\mathrm{E}}, \rightarrow)$, where "$\rightarrow$"denotes Lamport$'$s Happened Before Relation [10].

2.2 Background and Definitions

Happened Before Relation (HBR)

Definition 1. *The Happened Before Relation (HBR) [10], "$\rightarrow$", is the smallest relation on a set of events E, satisfying the following conditions:*

1. *If a and b are events belonging to the same process, and a was originated before b, then $a \rightarrow b$.*
2. *If a is the sending of a message by one process, and b is the delivery of the same message in another process, then $a \rightarrow b$.*
3. *If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.*

Immediate Dependency Relation (IDR) The IDR is the transitive reduction of the HBR. We denote it by "$\downarrow$", and it is defined as follows [7]:

Definition 2. *Two events $a, b \in E$ have an immediate dependency relation "$\downarrow$" if the following restriction is satisfied:*

$$a \downarrow b \text{ if } a \rightarrow b \text{ and } \forall c \in E, \neg(a \rightarrow c \rightarrow b)$$

Thus, an event a causal-immediately precedes an event b, if and only if no other event c belonging to E exists, such that c belongs to the causal future of a and to the causal past of b.

Quasi-Synchronous Checkpointing (QSC) QSC is a popular technique that can be used for fault-tolerance which allows processes to recover in spite of failures. Processes achieve fault-tolerance by asynchronously saving recovery information during their execution. When a failure occurs, previously saved recovery information (consistent global state) can be used to restart the computation from an intermediate state, therefore reducing the amount of lost computation. According to Netzer et al. [14], a local checkpoint can belong to a consistent global snapshot, which is composed of a set of local checkpoints C (one per process) if and only if no checkpoint in C has a Z-path with such local checkpoint (including itself). Even more restrictive, a local checkpoint that forms a Z-cycle cannot be part of any consistent global snapshot. Such kind of checkpoint is said to be useless and represents a waste of resources. These concepts are formally defined as follows.

Definition 3. *A communication and checkpoint pattern (CCP) is a pair $(\hat{E}, R_{\hat{E}})$ where $\hat{E}$ is a partially ordered set modeling a distributed computation, and $R_{\hat{E}}$ is a set of local checkpoints defined on $\hat{E}$. An example of communication and checkpoint pattern is given in Figure 2.*

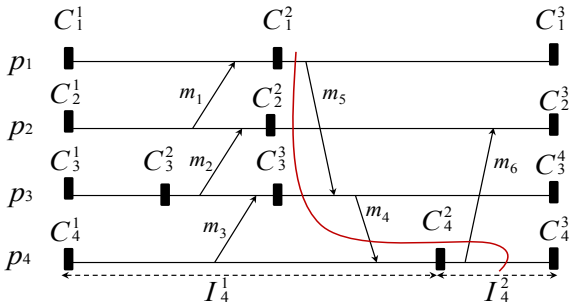


Figure 2: A communication and checkpoint pattern with a possible crash scenario

Netzer et al. [14] defined the notion of a zigzag path (Z-path) in a CCP as a generalization of HBR, as follows:

Definition 4. *A Z-path exists from C_i^x to another C_j^y ($C_i^x \xrightarrow{Z} C_j^y$) iff there are messages $m_1, m_2, ..., m_l$ such that:*

1. *m_1 is sent by process p_i after C_i^x,*

2. *if* m_k $(1 \leq k < l)$ *is received by process* p_r*, then* m_{k+1} *is sent by* p_r *in the same or at a later checkpoint interval (although* m_{k+1} *may be sent before or after* m_k *is received), and*

3. m_l *is received by process* p_j *before* C_j^y*.*

Helary et al. defined the following in [6].

Definition 5. *A Z-path* $[m_1, ..., m_l]$ *is causal, iff for each pair of consecutive messages* m_k *and* m_{k+1}*: delivery*$(m_k) \rightarrow$ *send*(m_{k+1})*. Otherwise, it is a non causal Z-path.*

Definition 6. *A non-causal Z-path from a checkpoint to a checkpoint is a sequence of messages* m_1*,* m_2*,* $\cdots$*,* m_n *satisfying the conditions of Definition 4, such that for at least one* i *(*$1 < i < n$*),* m_i *is received by some process Pr after sending the message* m_{i+1} *in the same checkpoint interval.*

Definition 7. *A Z-cycle is a noncausal Z-path from a local checkpoint* C_i^x *to itself:* $C_i^x \xrightarrow{Z} C_i^x$*.*

Definition 8. *The length l of a Z-path (or Z-cycle) is determined by the number of messages* m_1*,* m_2*, ...,* m_l *involved.*

In the above example (see Figure 2), the messages [m_5,m_4] form a causal Z-path of length two from the checkpoint C_1^2 to the checkpoint C_4^2. The messages [m_3,m_2,m_1] form a non-causal Z-path of length three from the checkpoint C_4^1 to the checkpoint C_1^2. Finally, the messages [m_4,m_3] form a Z-cycle at C_3^3 and [m_5,m_4,m_3,m_2,m_1] form a Z-cycle of length five at C_1^2. These patterns, as we will see, are automatically detected by our validation approach.

2.3 Classification of quasi-synchronous checkpointing

The main advantage of a QSC algorithm is that it can reduce the number of useless checkpoints. Quasi-synchronous checkpointing algorithms are classified into three different classes, namely, Strictly Z-Path Free (SZPF), Z-Path Free (ZPF), and Z-Cycle Free (ZCF) [12].

Strictly Z-path free checkpointing eliminates altogether all the noncausal Z-paths between checkpoints.

Definition 9. *A checkpointing pattern is said to be strictly Z-path free (or SZPF) if no noncausal Z-path exists between any two (not necessarily distinct) checkpoints.*

In a ZPF system, it is possible to prevent useless checkpoints by eliminating only those noncausal Z-paths in which there is no sibling causal path.

Definition 10. *A checkpointing pattern is said to be Z-path free (or ZPF) iff for any two checkpoints A and B, a Z-path exists from A to B iff a causal path from A to B exists.*

In a ZCF model, only Z-cycles are prevented.

Definition 11. *A checkpointing pattern is said to be Z-cycle free (or ZCF) iff none of the checkpoints lie on a Z-cycle.*

According to Netzer et al. [14] all checkpoints taken in SZPF, ZPF and ZCF systems are useful. The construction of consistent global states is more difficult in a ZCF than in the other systems because of the existence of noncausal Z-paths. However, a ZCF system has the potential to have the lowest checkpointing overhead as it takes forced checkpoints only to prevent Z-cycles. Based on the property ensured (SZPF, ZPF or ZCF), a QSC algorithm can guarantee that a set C of checkpoints generated in their solution achieves a consistent global snapshot. In other words, a QSC algorithm can guarantee a certainty evaluation of non-functional requirements only if it accomplishes the SZPF, ZPF or ZCF properties. From here, we note the importance of automatically detecting/validating if a quasi-synchronous checkpointing algorithm satisfies such properties, according to the case.

2.4 Graph Transformation

Graph Transformation is a rule-based approach targeted towards the modifications on a graph [17, 4, 3]. A rule is described by a pair of graphs $r = (L, R)$, where L is called the left-hand side graph and R is called the right-hand side graph. Applying the rule $r = (L, R)$ means finding a match of L in the host graph and replacing L by R. The suppression of the occurrence of L in G may cause the appearance of edges without a starting node or a terminating node or both. Those edges are called "dangling edges". Two main approaches have dealt with this problem, which are the SPO and the DPO approaches [3].

The Single PushOut Approach (SPO) A SPO rule is of the form (L, R). Its application to a graph G is related to the existence of an occurrence of L in G. The application of the SPO rule to a graph G involves the removal of the graph corresponding to $Del = (L \backslash (L \cap R))$ and the addition of an isomorphic copy of $Add = (R \backslash (L \cap R))$.

The Double PushOut Approach (DPO) A DPO rule is of the form (L, K, R), where K is used to clearly specify the invariant part to preserve after applying the rule. If both conditions of existence of the occurrence of L and absence of suspended edges are checked, the application of the rule involves the removal of the graph corresponding to the occurrence of $Del = (L \backslash K)$ and the addition of an isomorphic copy of $Add = (R \backslash K)$.

Neighborhood Controlled Embedding Approach (NCE) The NCE mechanism [17] is based on the specification of the connection instructions. Such instructions, described by a pair (n, δ), enrich rewriting rules by flexibly specifying the addition of edges. The execution of an instruction connection involves the introduction of an edge between the added node n and all the neighboring nodes of the removed node, whose label is δ. dNCE (d for edge direction), eNCE (e for edge label), and edNCE are extensions of the NCE approach. The dNCE connection instructions are described by a triplet (n, δ, d), where $d \in \{in, out\}$ controls the direction of the edges. The eNCE connection instructions are described by a triplet $(n, p/q, \delta)$, where p and q are edge labels. The edNCE is the combination of the eNCE and dNCE approaches. The edNCE connection instructions are of the form $(n; p/q; \delta; d; d')$. Such instructions lead to the

introduction of a q-labeled edge and in the direction indicated by d' between the node n and each node n' that used to be p-neighbours and d-neighbours (in-neighbours if d=in and out-neighbours otherwise) of a removed node.

3 Related Work

Finding a method to construct a consistent snapshot in a ZCF system has been an open problem. The impossibility of designing an optimal ZCF algorithm has been treated by Tsai et al. [20]. Recently, some algorithms which are ZCF have been proposed, for example, the Fully Informed (FI) algorithm of Helary et al. [6], the Fully Informed aNd Efficient (FINE) algorithm of Luo et al. [11], the Delayed Communication-Induced Checkpointing (DCFI) algorithm [18] and the Scalable Fully Informed (SF-I) algorithm [19] of Calixto et al. The present work introduces an approach for the validation of checkpointing properties that can be used with any CIC algorithm. To the best of our knowledge, it presents the first solution based on graph transformation accomplishing this purpose. Wang [21, 22] defines a graph called "*Rollback dependency graph*" to show Z-paths in a distributed computation. It is easy to detect Z-paths in such a graph, but detecting Z-cycles is a critical problem. Park et al. [15] propose a scheme for detecting Z-cycles of length two. Such scheme takes forced checkpoints to break them. Chin-Lin Kuo et al. [9] propose an in-line distributed algorithm to detect all Z-cycles of length more than two and their involved checkpoints. This algorithm conceptualizes an appropriate data structure to express Z-paths and Z-cycles. It requires considerable piggybacked Z-path information, but it detects, for a distributed computation, all the existing Z-cycles and their involved checkpoints. In order to use this last solution for validation purposes, such algorithm must be executed simultaneously at runtime along with the checkpointing algorithms, which implies an expensive and additional use of memory, processing and bandwidth resources.

4 Validation approach for checkpointing algorithms

Figure 3 illustrates the general process of our approach. To implement our approach, we have chosen the Graph Matching and Transformation Engine (GMTE)[1]. GMTE s a graph rewriting engine able to search small and medium patterns in huge graphs in a short time. As input it receives graph descriptions and transformation rules written in XML [5]. In our approach, the GMTE receives a checkpoint graph which models the execution of a checkpointing algorithm. The initial graph is modified using transformation rules to exhibit dangerous patterns or confirm their absence. In our solution, the specification of rewriting rules rely on both SPO and edNCE approaches. The SPO approach offers a flexible way of dealing with dangling edges, and it is known to be more expressive than the DPO approach. SPO rules are enriched with edNCE instruction to ease the addition and the removal of nodes.

[1]GMTE is available at http://homepages.laas.fr/khalil/GMTE/

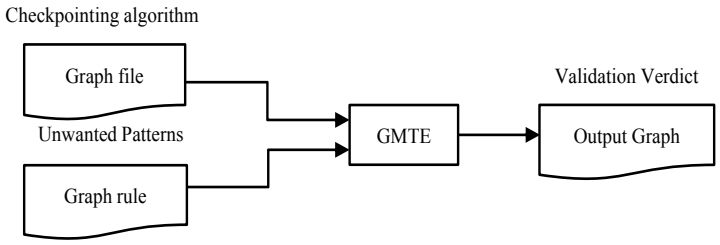


Figure 3: Validation process

4.1 Transformation rules for the validation of an HBR graph

In this section, we present a set of validation rules designed over the HBR graph. These rules were initially introduced in a previous work [8]. Figure 4 contains the HBR graph which corresponds to the scenario depicted in Figure 2.

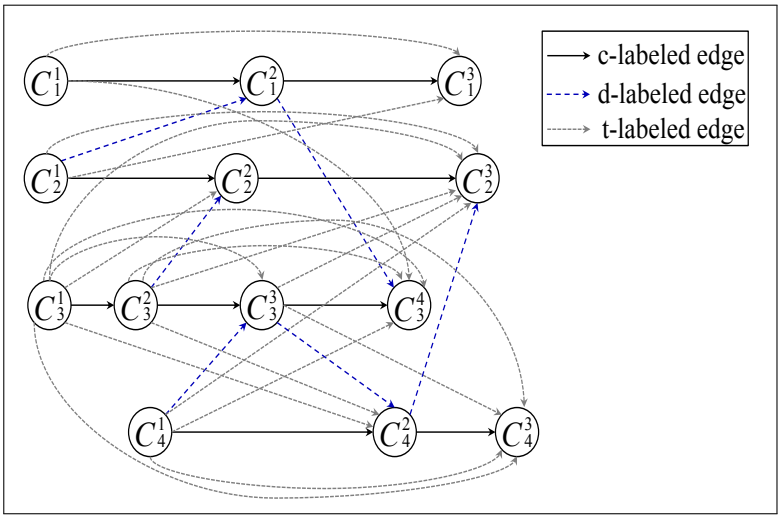


Figure 4: HBR graph of the scenario depicted in Figure 2

The HBR graph contains three types of edges: c-labeled edges for local relations, d-labeled edges for direct relations and t-labeled edges for transitive relations. The edges of an HBR graph give full causal information about a system execution; nevertheless, as we will see, working with the full graph considerably increases the complexity in finding the undesirable patterns.

Figure 5 shows the general patterns of Z-paths and Z-cycles in HBR and IDR graphs. In an HBR graph, a Z-path can be of two forms depending on its length. A Z-path of length two is a pattern ($n_1 \xrightarrow{d} n_3$, $n_2 \xrightarrow{c} n_3$, $n_2 \xrightarrow{d} n_4$), as shown in Figure 5.(a). A long Z-path (i.e., a length greater than two) is, as illustrated in Figure 5.(d), a succession of Z-paths of length two. Similarly, a Z-cycle is either of length two and in the form ($n_1 \xrightarrow{d} n_3$, $n_2 \xrightarrow{c} n_3$, $n_2 \xrightarrow{d} n_1$) (see Figure 5.(b)) or is a succession of Z-paths from one checkpoint to itself (see

Figure 5.(e)).

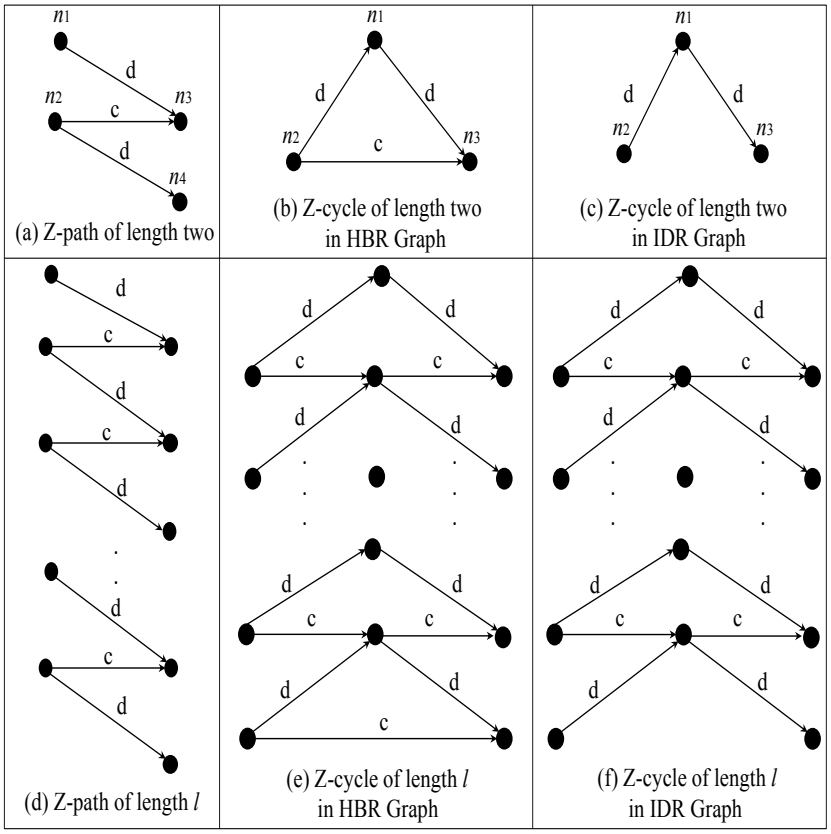


Figure 5: General patterns of Z-paths and Z-cycles

Firstly, we check whether the HBR graph is exempt from Z-paths of length two using rule r_1 characterized in Figure 6. The application of r_1 leads to the addition of a z-labeled edge between two checkpoints (nodes) involved in a Z-path of length two. The application of r_1 to G_1 shows the existence of three Z-paths: from C_1^2 to C_4^2 ($C_1^2 \xrightarrow{d} C_3^4$, $C_3^3 \xrightarrow{c} C_3^4$, $C_3^3 \xrightarrow{d} C_4^2$), from C_4^1 to C_2^2 ($C_4^1 \xrightarrow{d} C_3^3$, $C_3^2 \xrightarrow{c} C_3^3$, $C_3^2 \xrightarrow{d} C_2^2$) and from C_3^2 to C_1^2 ($C_3^2 \xrightarrow{d} C_2^2$, $C_2^1 \xrightarrow{c} C_2^2$, $C_2^1 \xrightarrow{d} C_1^2$). Rule r_2 characterized in Figure 7 detects long Z-paths. Two successive Z-paths form another Z-path ($n_1 \xrightarrow{z} n_3$, $n_2 \xrightarrow{d} n_3$, $n_2 \xrightarrow{z} n_4$). Detected Z-paths are once again exhibited by adding a z-labeled edge. Figure 7 illustrates the application of r_2 to G_2, giving G_3. Here, there exists a long Z-path from C_4^1 to C_1^2 ($C_4^1 \xrightarrow{z} C_2^2$, $C_3^2 \xrightarrow{d} C_2^2$, $C_3^2 \xrightarrow{z} C_1^2$).
The second verification step of our approach tackles the detection of Z-cycles. It is conducted through the sequential execution of the rules r_3 and r_4. Rule r_3 (presented in Figure 8) is used to detect Z-cycles of length two. During its application, any vertex involved in a Z-cycle is replaced by a zc-labeled node. Figure 8 illustrates the application of r_3 to G_3, resulting in G_4. In the running example, C_3^3 is involved in the only existing Z-cycle of length two. In turn, the rule r_4 allows the detection of long Z-cycles. Figure 9 presents r_4 and

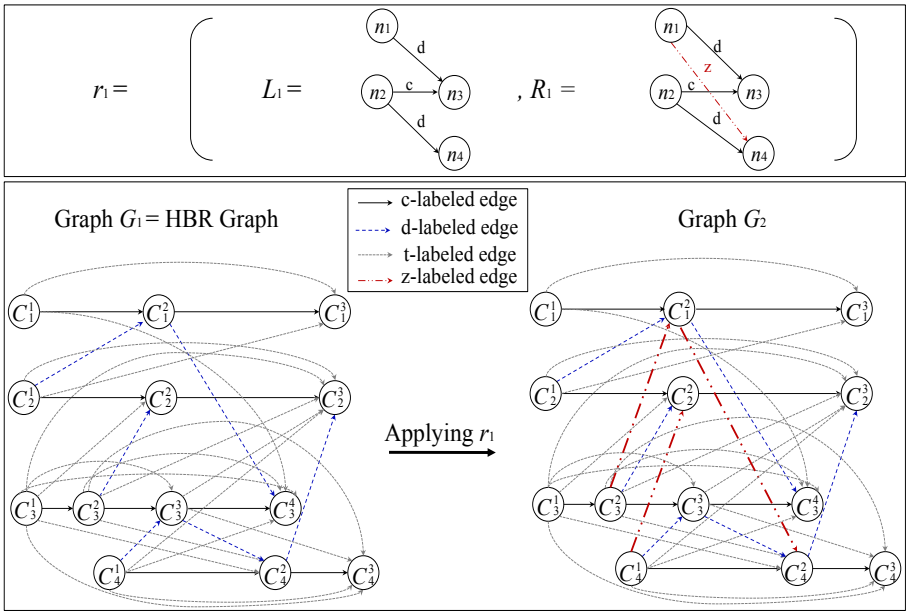


Figure 6: Application of rule r_1

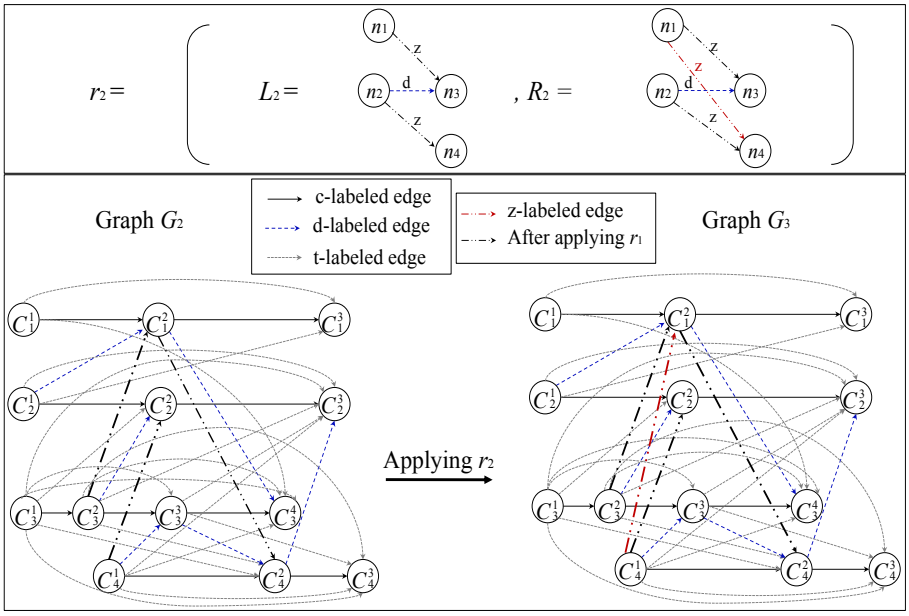


Figure 7: Application of rule r_2

illustrates its application on G_4. By applying rule r_4, each node involved in a long Z-cycle is replaced by a new zc-labeled node. In the given example, a z-cycle involving C_1^2 ($C_1^2 \xrightarrow{z} C_4^2$, $C_4^1 \xrightarrow{c} C_4^2$, $C_4^1 \xrightarrow{z} C_1^2$) exists. In both rules, connection instructions are used to preserve vertices; For any edge e such as

n_1 (the deleted vertex), a similar vertex with the zc-labeled vertex (the added vertex) is added. The result of the transformation process is a graph whose Z-paths and Z-cycles have been put under the spotlight. We therefore can decide whether the algorithm is ZPF, ZCF or none of these. In the running example, the graph Output G_5 shows the existence of Z-paths and Z-cycles. Thus, this system is neither ZPF nor ZCF.

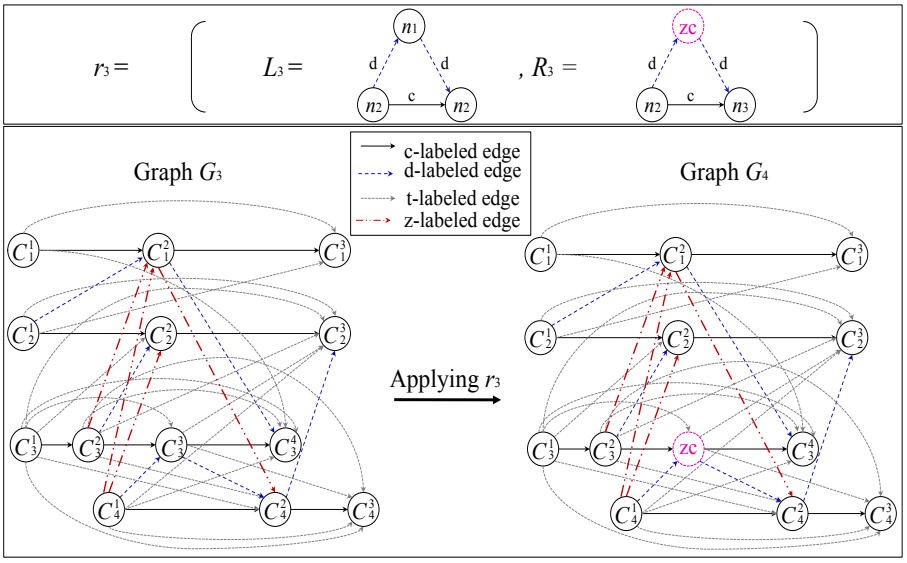


Figure 8: Application of rule r_3

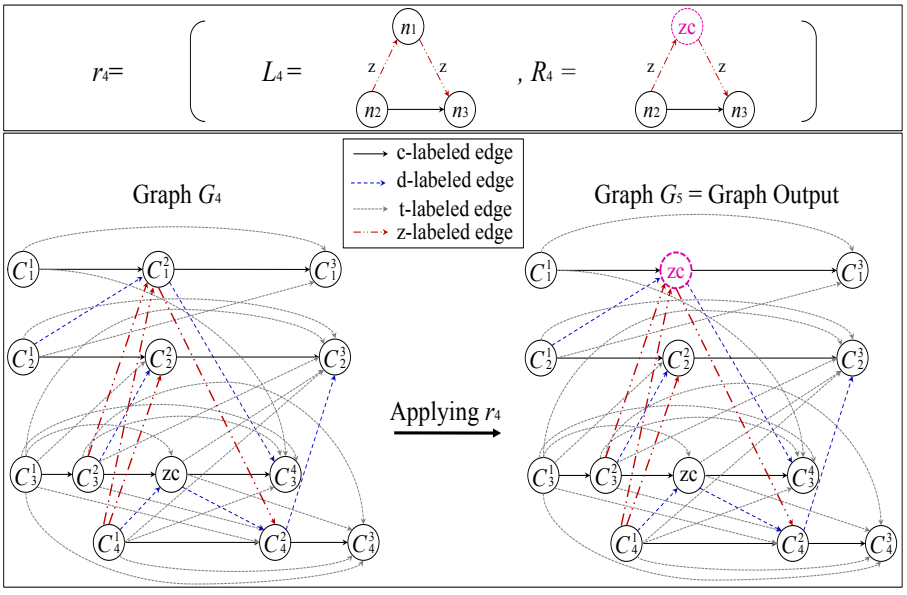


Figure 9: Application of rule r_4

4.2 Transformation rules for the validation of a IDR graph

By applying the HBR rules to the HBR graph, we can detect all existing Z-paths and Z-cycles regardless of their length. However, the application of these rules becomes more time-consuming on big graphs. In fact, one main problem linked to the Happened Before Relation is the combinatorial state explosion [1] which exponentially increases with the number of processes. To lift this limitation and for efficiency sake, we propose the study of the minimal causal graph (IDR graph). To this end, we implemented the Minimal Causal and Set Representation (MCSR)[2] tool which constructs, for a distributed computation, the corresponding IDR graph. Such a graph constitutes a significant reduction in the state-space of the system. The IDR graph presented in Figure 10 corresponds to the scenario depicted in Figure 2. Compared with the HBR graph, the IDR graph does not contain transitive edges (all t-labeled edges and some c-labeled edges, which are local and transitive at the same time, are deleted).

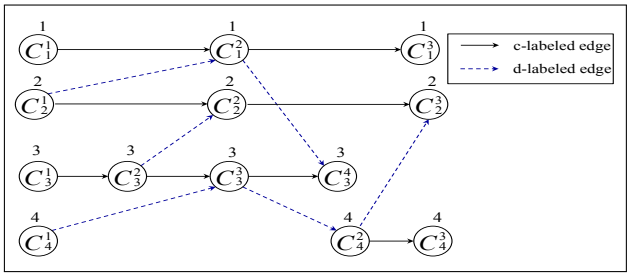


Figure 10: Example of an IDR graph

Similarly, we present the general pattern of a Z-path (Figure 5. (a) and (d)) and a Z-cycle (Figure 5. (c) and (f)) in an IDR graph. We remark that some c-labeled edges, which are transitive, are removed, thereby making the detection of Z-cycles more difficult. To solve this issue, each vertex of the IDR graph is labeled by the number of the process to which it belongs. By doing so, detecting Z-paths in IDR and HBR graphs can be done in the same way, through the execution of r_1 and r_2. The detection of Z-cycles in IDR graphs, however, requires new rules described below.

We start by applying rules r_1 and r_2 to detect Z-paths. Their application to the IDR graph of Figure 10 exhibits the existence of three Z-paths of length two from C_1^2 to C_4^2, from C_4^1 to C_2^2, and from C_3^2 to C_1^2 and of a long Z-path of length three from C_4^1 to C_1^2 (see Figure 11).

In general, a Z-cycle of length two in an IDR graph is in the form of ($n_1 \xrightarrow{d} n_2$, $n_2 \xrightarrow{d} n_3$) as described in Figure 5. (c). However, this pattern does not necessarily imply a Z-cycle; it does if and only if n_1 and n_3 belong to the same process. To determine whether a minimal Z-cycle exists indeed, rule r3 relabels the first edge of such a pattern. The value of this new label is equal to the difference between n_1 and n_3's process number. If the result is 0 (i.e., the edge is 0-labeled), n_1 and n_3 belong to the same process and the edge is part of a minimal Z-cycle. The application of r_3 to the running example G_2 is depicted in Figure 12. We note the existence of an 0-labeled edge indicating a Z-cycle

[2]MCSR is available at http://homepages.laas.fr/khalil/GMTE/

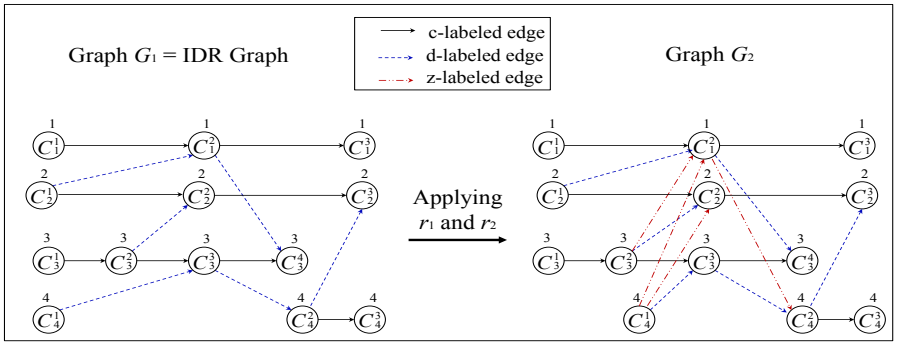


Figure 11: Application of r_1 and r_2

of length two involving $C_3^3$and a 1-labeled edge corresponding to the successive d-labeled edges ($C_3^3 \xrightarrow{z} C_4^2$, $C_4^2 \xrightarrow{d} C_2^3$) which do not belong to a Z-cycle.

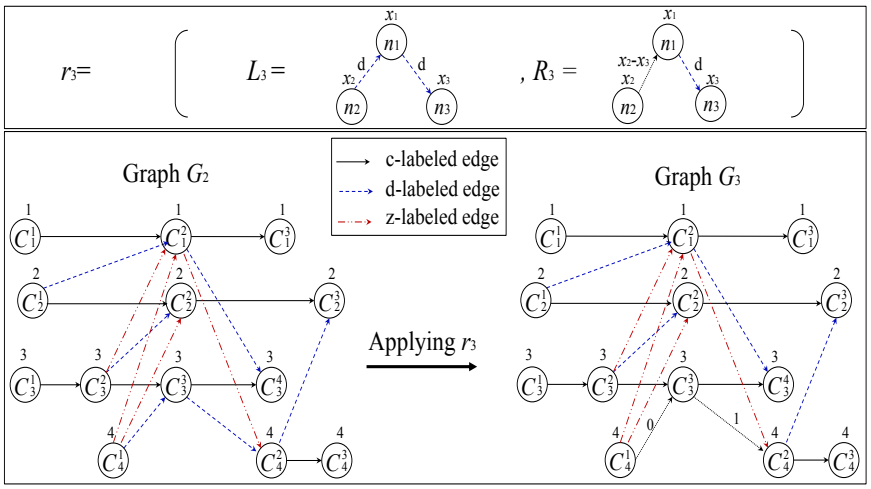


Figure 12: Application of r_3

Rule r_4 defined in Figure 13 exploits this new representation. It detects 0-labeled edges and replaces their destination vertex by a zc-labeled, revealing the existence of a Z-cycle pattern. Its application to the running example emphasizes the existence of a Z-cycle involving C_3^3 (see Figure 13). This node is replaced by a zc-labeled node. As we have previously mentioned, the removal and the addition of nodes requires the definition of connection instructions to retrieve all existing connections. Finally, rule r_5, specified in Figure 14, detects long Z-cycles. A long Z-cycle, in an IDR graph, is in the form of ($n_1 \xrightarrow{z} n_2$, $n_2 \xrightarrow{z} n_3$). Rule r_5 is applied to G_4. Its application detects a long Z-cycle involving C_1^2. The output graph G5 of the IDR process shows all existing Z-cycles. Therefore, by applying the IDR rules, we can also deduce whether the system is ZPF or ZCF. This is formally proved in Section 5.

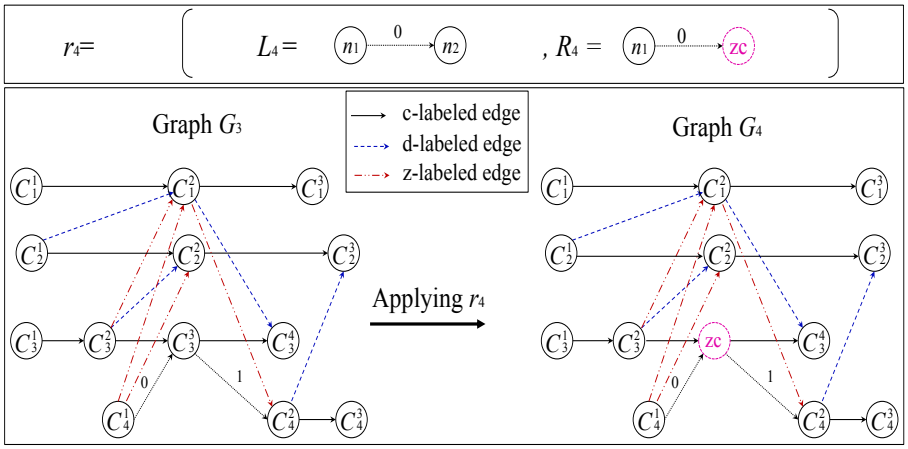


Figure 13: Application of r_4

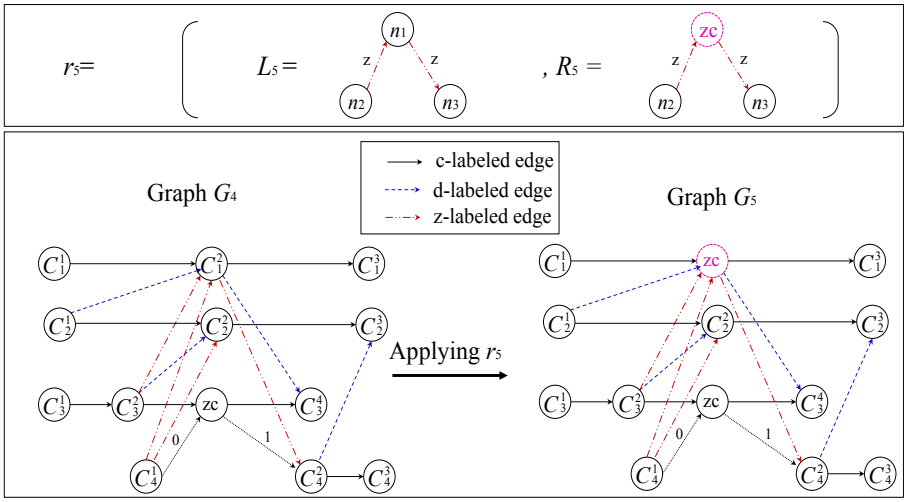


Figure 14: Application of r_5

4.3 Interpreting the result

We focus on the Z-cycles patterns detected in G_5 for HBR and IDR, since these patterns are also composed of Z-paths. We recall that, in order to construct a consistent global state we need to find a set C of checkpoints (one by each process), which are Z-path free among them. In order to guarantee the non-functional requirement of fault-tolerance in the scenario depicted in Figure 2, we need to establish the nearest point of recovery (consistent global state) before the crash. In this scenario, in a first attempt to establish such point of recovery C, we take the checkpoints C_1^2, C_2^2 and C_3^3 since they are Z-path free among them. Then, we verify if C_4^1 or C_4^2 can be included in C. According to the graph G_5 , the Z-cycles at C_3^3 and C_1^2 prevent both C_4^1 and C_4^2 from being included in C. And as consequence, it is not possible to construct a consistent snapshot with such checkpoints.

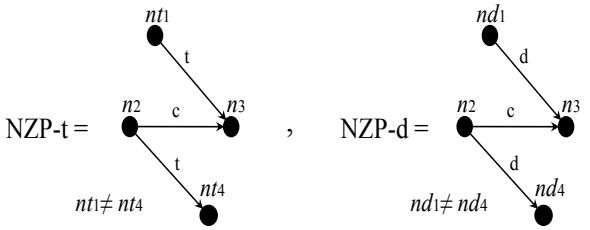
Understanding why. To be consistent a global snapshot must contain all the events in its causal history. In this case, if we take C_4^1 and C_3^3, the checkpoint C_3^3 knows about the *delivery* of m_3 but at C_4^1 the *send* of m_3 is missing. The same situation occurs between C_3^3 and C_4^2 with respect to m_4 in an opposite way. For these reasons C_4^1 and C_4^2 cannot be included in C.

5 Formal proof

Previously, we have presented a set of patterns to detect Z-paths and Z-cycles, in the first time it was in the HBR graph, and in the second time it was in the IDR graph. In this section, we explain why we have chosen such patterns and why these patterns are able to detect Z-paths and Z-cycles in such graphs.

Z-path Proof Firstly, we recall the definition of a noncausal Z-path. A noncausal Z-path from a checkpoint to a checkpoint is a sequence of messages, provided that at least two successive messages m_i and m_{i+1} exist such that delivery(m_{i+1}) $\rightarrow$ send(m_i). Then, we can assume two possible cases. In the first case, the noncausal Z-path contains only two noncausal messages. In the second case, all the messages are sent in a noncausal way, which corresponds to the most complex case.
In Figure **??**, we present the general pattern of noncausal Z-paths in the HBR and IDR graphs. We note two main forms in the HBR graph, which are *t-c-t* and *d-c-d*. Then, as the IDR graph does not contain transitive edges, we note only the form *d-c-d* in such a graph.
We proof the correspondence between the patterns in the HBR and IDR graphs, as follows:

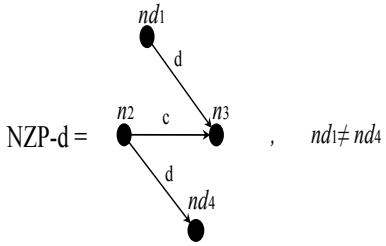
Condition 1. *A Quasi-synchronous checkpointing algorithm is ZPF if its HBR graph G = (R, $\rightarrow$) does not contain the two following subgraphs:*

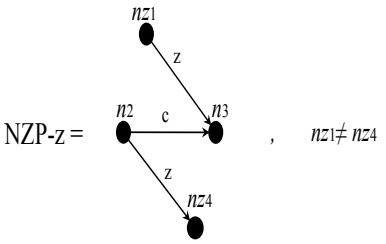


Condition 2. *A Quasi-synchronous checkpointing algorithm is ZPF if its IDR graph G′ = (R, $\downarrow$) does not contain the following subgraph:*

Theorem 1. *Condition 2 is equivalent to Condition 1.*

Proof. Before presenting the proof, we need to make some assertions. We modeled a non causal Z-path of length l, in the form of:
where the arc z can be a set of nested subgraphs of the form NZP-t and/or NZP-d. We note that, since the IDR is the transitive reduction of the HBR, we have $G' \subseteq G$ with the same transitive closure. Keeping this in mind, we divide

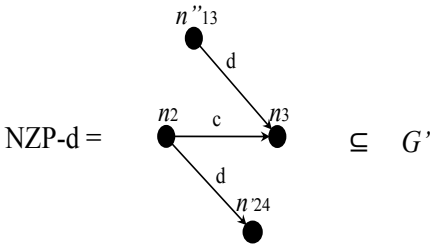




the proof into two main parts. In the first part, we focus on demonstrating that for each NZP-t pattern in G we are able to find a NZP-d pattern in G'. In the second part, we demonstrate that for each NZP-Z pattern in G we are able to find a NZP-d in G'.
For the first part, we define the following Lemma:

Lemma 1. *If a NZP-t $\subseteq G$ exists, then a NZP-d $\subseteq G'$ exists such that NZP-d $\subset$ NZP-t.*

We proof in a direct form. For the pair $nt_1 \xrightarrow{t} n_3$ and $n_2 \xrightarrow{t} nt_4$, it means that, a sequence of events (nodes) exists in G', between nt_1 and n_3, such that $nt_1 \xrightarrow{d} n'_{13} \xrightarrow{d} ... \ n''_{13} \xrightarrow{d} n_3$, and between n_2 and nt_4 of the form $n_2 \xrightarrow{d} n'_{24}$ $\xrightarrow{d} ... \ n''_{24} \xrightarrow{d} nt_4$. Since the pair $n_2 \xrightarrow{c} n_3$ belongs to both graphs G and G' we have that the following pattern:

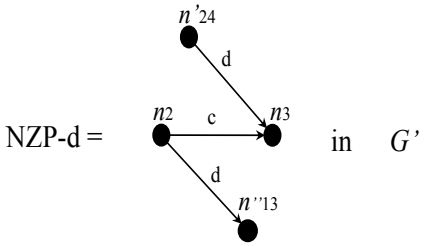


It is included in the NZP-t $\subseteq G$.

Lemma 2. *If a NZP-z $\subseteq G$ exists, then a NZP-d $\subseteq G'$ exists such that NZP-d $\subset$ NZP-z.*

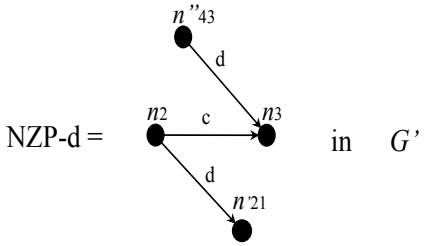
We proof in a direct form. We have three cases according to it: the arc z represents a NZP of the form NZP-d, NZP-t or nested. For simplicity, but without a loss of generality, we assume that the two arcs z in the NZP-Z are of the same type.

Case 1 NZP-z = NZP-d This case implies that a node exists between nz_1 and n_3 and another node between n_2 and nz_4, such that $n'_{24} \xrightarrow{d} n_3$ and $n_2 \xrightarrow{d} n''_{13}$, respectively. Since the pair $n_2 \xrightarrow{d} n_3$ belongs to both graphs G and G' we have the following pattern:



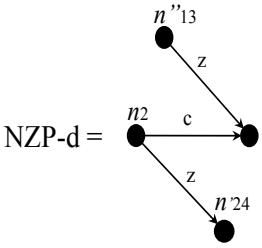
It is included in the NZP-t $\subseteq G$.

Case 2 NZP-z = NZP-t This case implies that a node exists between nz_1 and n_3, and another node between n_2 and nz_4 such that $nt_4 \xrightarrow{t} n_3$ and $n_2 \xrightarrow{t} nt_1$, respectively. By using Lemma 1, this means that, a finite sequence of nodes exists such that, $nt_4 \xrightarrow{d} n'_{43} \xrightarrow{d} ... \; n''_{43} \xrightarrow{d} n_3$ and $n_2 \xrightarrow{d} n'_{21} \xrightarrow{d} ... \; n''_{21} \xrightarrow{d} nt_4$. Since the pair $n_2 \xrightarrow{c} n_3$ belongs to both graphs G and G' we have the following pattern:



It is included in the NZP-t $\subseteq G$

Case 3 Nested NZP This case implies that a finite set of NZP patterns exists between nz_1 and n_3, and n_2 and nz_4 in the form $nz_1 \xrightarrow{z} n'_{13} \xrightarrow{z} ... \; n''_{13} \xrightarrow{z} n_3$ and $n_2 \xrightarrow{z} n'_{24} \xrightarrow{z} ... \; n''_{24} \xrightarrow{z} nz_4$. Since the pair $n_2 \xrightarrow{c} n_3$ belongs to both graphs, G and G' we identify the following single pattern:



It is included in the NZP-t $\subseteq G$.

From this result, it follows to apply either Case 1 or Case 2 according to the type of z arcs.

□

Z-cycle Proof By definition, a Z-cycle is a noncausal Z-path from a local checkpoint to itself. Causal Z-cycles cannot exist because an event cannot happen before itself. In Figures 15 and 16, we illustrate the general pattern of a Z-cycle in HBR and IDR graphs. We consider three cases of a Z-cycle: a Z-cycle of length two and two cases of a long Z-cycle. In the first case, the Z-cycle contains only two noncausal messages. In the second case, all messages belonging to the Z-cycle are sent in a noncausal way, which corresponds to the most complex case.

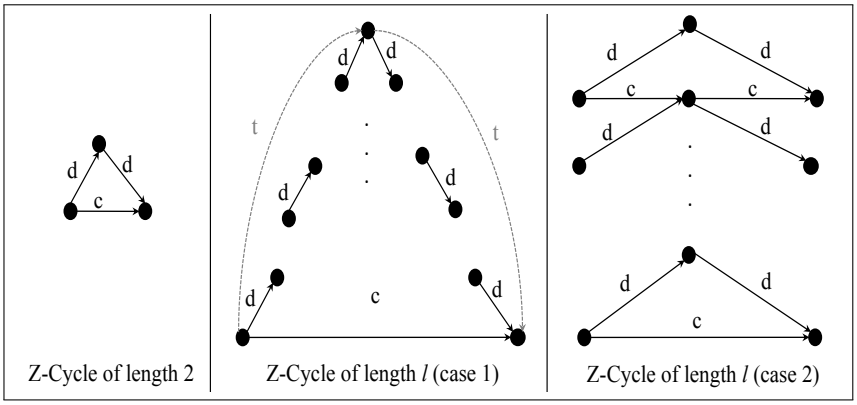


Figure 15: General pattern of a Z-cycle in HBR graph

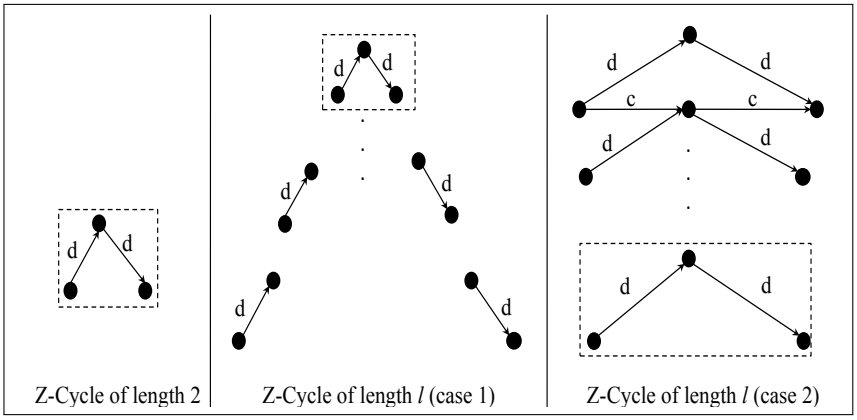
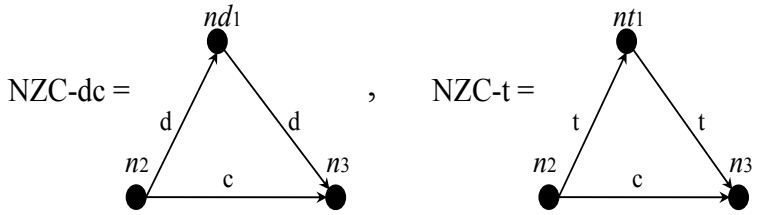
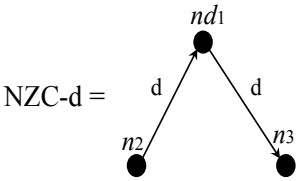


Figure 16: General pattern of a Z-cycle in IDR graph

Condition 3. *A Quasi-synchronous checkpointing algorithm is ZCF if its HBR graph $G = (R, \rightarrow)$ does not contain the two following subgraphs:*

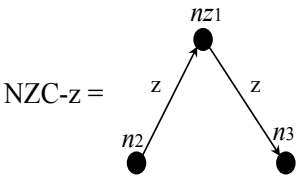


Condition 4. *A Quasi-synchronous checkpointing algorithm is ZCF if its IDR graph $G' = (R, \downarrow)$ does not contain the following subgraph:*



Theorem 2. *Condition 4 is equivalent to Condition 3.*

The case of a Z-cycle of length l, we modeled in the form of:



Proof. We divide the proof into two main parts. In the first part, we focus on demonstrating that for each NZC-t pattern in G, we are able to find a NZC-d pattern in G'. In the second part, we demonstrate that for each NZC-z pattern in G, we are able to find a NZP-d in G'.

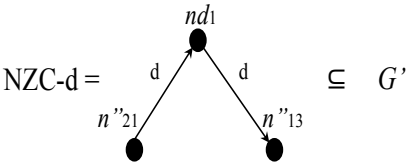
For the first part, we define the following Lemma:

Lemma 3. *If a NZC-t $\subseteq G$ exists, then a NZC-d $\subseteq G'$ exists such that NZC-d $\subset$ NZC-t.*

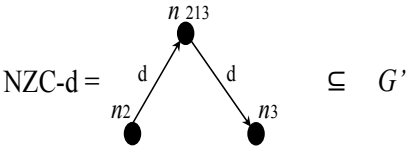
We proof in a direct form. For the pair $nt_1 \xrightarrow{t} n_3$ and $n_2 \xrightarrow{t} nt_1$, a sequence of events (nodes) in G' exists between nt_1 and n_3, such that $nt_1 \xrightarrow{d} n'_{13} \xrightarrow{d}$ ... $n''_{13} \xrightarrow{d} n_3$ and between n_2 and nt_1 of the form $n_2 \xrightarrow{d} n'_{21} \xrightarrow{d}$ ... $n''_{21} \xrightarrow{d}$ nt_1. Since the pair $n"_{21}$ and $n"_{13}$ belongs to both graphs G and G' we have the following pattern:
It is included in the NZC-t $\subseteq G$.

Lemma 4. *If a NZC-z $\subseteq G$ exists, then a NZC-d $\subseteq G'$ exists such that NZC-d $\subset$ NZC-z.*

Likewise, we have three cases according to it: the arc z represents a NZC of the form NZC-d, NZC-t or nested.

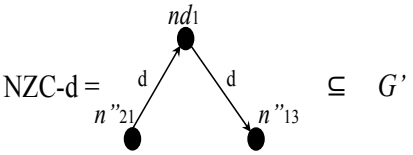


Case 1 NZC-z = NZC-d This case implies that a node exists between nz_1 and n_3, and between n_2 and nz_1, such that $n_{213} \xrightarrow{d} n_3$ and $n_2 \xrightarrow{d} n_{213}$, respectively. Since the pair n_2 and n_3 belongs to both graphs G and G' we have the following pattern:



It is included in the NZC-t $\subseteq G$.

Case 2 NZC-z = NZC-t This case implies that a node exists between nz_1 and n_3, and another node between n_2 and nz_1, such that $nt_1 \xrightarrow{t} n_3$ and $n_2 \xrightarrow{t} nt_1$, respectively. By using Lemma 3, this means that a finite sequence of nodes exists such that $nt_1 \xrightarrow{d} n'_{13} \xrightarrow{d} ... \ n''_{13} \xrightarrow{d} n_3$ and $n_2 \xrightarrow{d} n'_{21} \xrightarrow{d} ... \ n''_{21} \xrightarrow{d} nt_1$. Since the pair $n"_{21}$ and $n"_{13}$ belongs to both graphs G and G' we have the following pattern:



It is included in the NZC-t $\subseteq G$.

Case 3 Nested NZC This case implies that a finite set of NZP patterns exists between nz_1 and n_3, and n_2 and nz_1 in the form $nz_1 \xrightarrow{z} n'_{13} \xrightarrow{z} ... \ n''_{13} \xrightarrow{z} n_3$ and $n_2 \xrightarrow{z} n'_{21} \xrightarrow{z} ... \ n''_{21} \xrightarrow{z} nz_1$. This means that, it follows to apply Case 1 or Case 2 according to the type of z-labeled arcs.

□

6 Performance analysis

The objective of this section is to illustrate the efficiency of designing the IDR rules. As we have previously mentioned, the IDR graph is the transitive reduction of the HBR graph. The use of such graph to validate checkpointing algorithms is an important contribution in terms of cost reduction. In fact, the

cost of graph transformation approaches depends on the number of nodes and edges in the host graph. In general, there are at maximum $max(CO_q^t, CO_k^m)$ occurrences of the pattern L in the host graph G, where t= number of nodes in L, m= number of edges in L, q= number of nodes in G and k= number of edges in G (CO_r^l gives the number of possible combinations of l objects from a set of r objects).
Let p and n be two integers (p = number of processes in the system, n = number of events). The maximum number of edges in the HBR graph and the IDR graph is when we have the maximum number of concurrent messages in each round (see Figure 17). Therefore, the number of nodes is n = number of rounds $\times$ p.

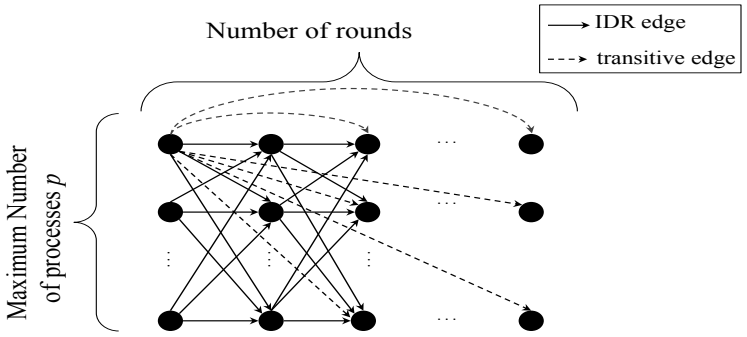


Figure 17: Maximum number of edges in an HBR graph

The maximum number of edges in the HBR graph (HBR edges = IDR edges + transitive edges) is:

$$e = p(n\text{-}p) + \frac{(n-p)(n-2p)}{2}$$

The maximum number of edges in the IDR graph is only:

$$e' = p(n\text{-}p)$$

Figure 18 shows the maximum number of matches used to detect Z-paths of length two in HBR and IDR graphs. These graphs present three executions of a system with 100 processes and a different number of nodes (500, 1000, 2000, 3500, and 5000, respectively). The Z-path of length two is composed of 4 nodes and 3 edges. Consequently, the maximum number of matches is max(CO_n^4, CO_e^3) if this pattern is detected in the HBR graph, and it is max(CO_n^4, $CO_{e'}^3$) in case of an IDR graph. The results show that both the HBR and the IDR curves are increasing. The number of matches (three edges (Z-path) of e edges (in HBR graph) or e' edges (in IDR graph)) increases when the number of nodes increases, and it is much higher in the HBR curve. To conclude, the execution of the IDR rules not only detects the same number of dangerous patterns as the HBR rules but also provides a considerable cost reduction.

7 Conclusion and future work

The need for checkpointing is intensified with the occurrence of autonomic computing systems. By saving local states periodically and determining a consistent global state, checkpointing techniques make the system self-aware about

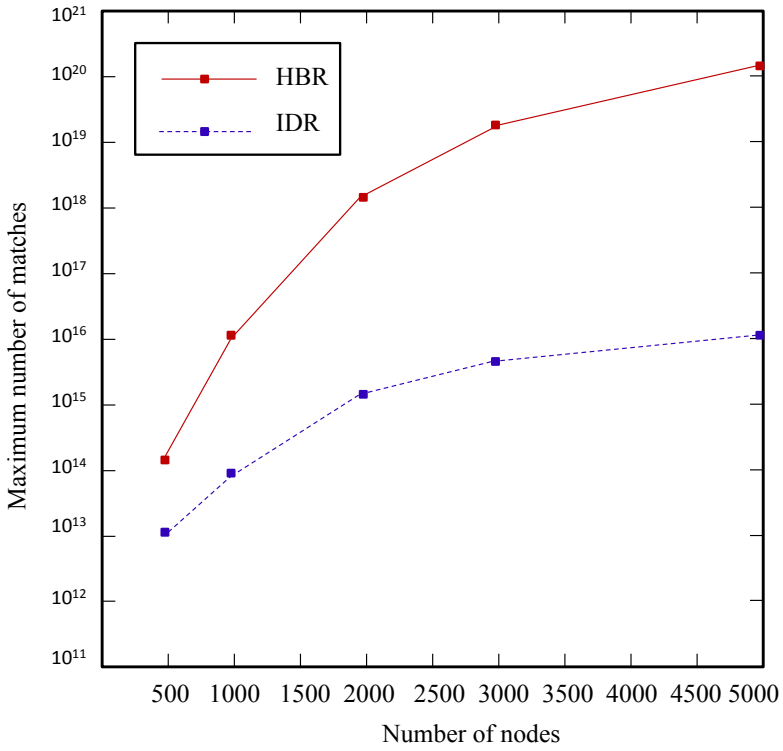


Figure 18: Comparison of number of matches in HBR and IDR graphs

its current state and enable it to repair or reconfigure itself in case of failures. To construct a consistent global state, quasi-synchronous checkpointing algorithms detect dangerous checkpointing patterns, such as Z-paths and Z-cycles and break them by taking forced checkpoints.

In this paper, we have proposed a validation approach for quasi-synchronous checkpointing algorithms. We have used graph transformation approaches to verify the correctness of such algorithms. To achieve this, we have modeled the system execution by first using the HBR graph and then by using the IDR graph. Then, we designed a set of transformation rules to verify if the algorithm is exempt from non-desirable patterns in such graphs. The application of these rules shows all existing Z-paths and Z-cycles regardless of their length. With the obtained results, we can validate that the algorithm is ZPF and determine whether it is ZCF. The use of graph transformation approaches ensures correct results without much need of time and space. The low cost is explained by the fact that the designed rules are not executed at runtime. Added to that, the use of the IDR rules ensures a cost reduction compared with the HBR rules.

As a future work, we aim to design transformation rules oriented to the Minimal Causal and Compact Graph (CAOS graph). This graph is based on the Causal Order Set Abstraction (CAOS) to present causal dependencies in terms of sets of events. Compared with the IDR graph, it greatly reduces the number of nodes and edges. This is the reason why finding an efficient method to detect non desirable patterns in such graph can be an important contribution in terms

of cost reduction.

References

[1] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(3):279–287, 1999.

[2] M. O. Cordier, Y. Pencol. T. Massuy and T. Vidal. Characterizing and checking self-healability. In *Proceedings of the 2008 conference on ECAI 2008: 18th European Conference on Artificial Intelligence*, pages 789–790. IOS Press Amsterdam, The Netherlands, 2008.

[3] H. Ehrig, H. Kreowski, and G. Rozenberg. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In *Graph-Grammars and Their Application to Computer Science*, volume 532 of *LNCS*. Springer, March 5-9 1990.

[4] J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In *Handbook of Graph Grammars*, pages 1–94. World Scientific Publishing, 1997.

[5] M. A. Hannachi, I. Bouassida-Rodriguez, K. Drira, and S. P. Hernandez. GMTE: A tool for graph transformation and exact/inexact graph matching. In *Graph-Based Representations in Pattern Recognition. 9th IAPR-TC-15 International Workshop, GbRPR 2013, Vienna, Austria*, volume 7877 of *LNCS*. Springer, 2013.

[6] J. M. Helary, A. Mostefaoui, R. H. B. Netzer, and M. Raynal. Communication-based prevention of useless checkpoints in distributed computations. *Distributed Computing*, 13:29–43, January 2000.

[7] S. E. P. Hernandez. The minimal dependency relation for causal event ordering in distributed computing. *Applied Mathematics & Information Sciences*, 9(1):57–61, Janury 2015.

[8] H. Khlif, H. Hadj-Kacem, S. P. Hernandez, C. Eichler, A. Hadj-Kacem, and A. Simon. A graph transformation-based approach for the validation of checkpointing algorithms in distributed systems. In *Proceedings of the 23nd IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 80–85, 2014.

[9] C. L. Kuo and Y. M. Yeh. An algorithm for detecting z-cycles in distributed computing system. In *Int. Computer Symposium*, pages 1124–1133, December 2004.

[10] L. Lamport. Time, clocks and the ordering of events in a distributed system. *ACM*, 21:558–565, July 1978.

[11] Y. Luo and D. Manivannan. Fine: A fully informed and efficient communication-induced checkpointing protocol for distributed systems. *Journal of Parallel and Distributed Computing*, 69:153–167, February 2009.

[12] D. Manivannan and M. Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713, 1999.

[13] R. H. B. Netzer and J. Xu. Adaptive message logging for incremental replay of message-passing programs. Technical report, Brown University, Providence, RI, USA, 1993.

[14] R. H. B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6:165–169, February 1995.

[15] T. Park and H. Y. Yeom. Application controlled checkpointing coordination for fault-tolerant distributed computing systems. *Parallel Computing*, 26:467–482, 2000.

[16] Y. Pencole. Diagnosability analysis of distributed discrete event systems. In *16th European Conference on Artificial Intelligence ECAI 2004, August 22-27, Valencia, Spain*, pages 43–47. IOS Press, 2004.

[17] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*, volume 3. World Scientific Publishing Co., Inc., 1997.

[18] A. C. Simon, S. E. P. Hernandez, and J. R. P. Cruz. A delayed checkpoint approach for communication-induced checkpointing in autonomic computing. In *Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 2013.

[19] A. C. Simon, S. E. P. Hernandez, J. R. P. Cruz, P. Gomez-Gil, and K. Drira. A scalable communication-induced checkpointing algorithm for distributed systems. *IEICE Transactions on Information and Systems*, E96-D(4):886–896, April 2013.

[20] J. Tsai, Y. Wang, and S. Kuo. Evaluations of domino-free communication-induced checkpointing protocols. *Information Processing Letters*, 69:1–69, 1998.

[21] Y. M. Wang. Maximum and minimum consistent global checkpoints and their applications. In *Symposium on Reliable Distributed Systems*, pages 86–95, 1995.

[22] Y. M. Wang. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, April 1997.