Towards a Concurrence Analysis in Business Processes

Anastasija Nikiforova
Faculty of Computing
University of Latvia
Riga, Latvia
Anastasija.Nikiforova@lu.lv

Janis Bicevskis
Faculty of Computing
University of Latvia
Riga, Latvia
Janis.Bicevskis@lu.lv

Girts Karnitis
Faculty of Computing
University of Latvia
Riga, Latvia
Girts.Karnitis@lu.lv

***Abstract*- This paper presents first steps towards a solution aimed to provide concurrent business processes analysis methodology for predicting the probability of incorrect business process execution. The aim of the paper is to (a) look at approaches to describing and dealing with the execution of concurrent processes, mainly focusing on the transaction mechanisms in database management systems, (b) present an idea and a preliminary version of an algorithm that detects the possibility of incorrect execution of concurrent business processes. Analyzing business process according to the proposed procedure allows to configure transaction processing optimally.**

Keywords— business process correctness, concurrence control, symbolic execution, transaction.

I. Introduction

Nowadays, the majority of the systems support multiple users' simultaneous access. Any user should have a possibility to access the system or database (DB) without any concern regarding other users that can modify the same data at the same time. This means, that very efficient concurrency managing mechanisms should be involved, that would ensure the impression that all the operations are executed in such a way that they are executed serially – one by one. If user(s) requests require updating the data (insert, update or delete), special algorithms are required for concurrent writings. The problems that might arise relate to cases where multiple users send requests for modifying the same data set, or some of them send updating requests and others send retrieval requests. The most common way to solve these challenges is to block access to the DB while each request is resolved. In this way, each request is resolved sequentially, but the operation of the system is affected significantly.

Numerous solutions were carried out in recent decades, some of which will be discussed in this paper, taking into account the main aspects and areas to which they relate. Despite users' perspective, it can be assumed that this challenge has been already resolved and that there is a list of possible alternatives already in-built in database management systems (DBMS), recent studies have shown that this issue is still ongoing even in the scope of leading DBMS (for instance, [1]). In addition, relatively new topics such as Internet of Things (IoT) [2]-[5] and even metadata performance scalability [6] deal with concurrent processes. Moreover, recent studies have indicated that this topic is important even in the case of analytical and modeling solutions combating COVID-19 [7]. However, this study proposes a completely different solution that is not limited to specific DBMS, focusing on modeling the concurrent business processes to ensure that the probability of incorrect concurrent execution is low.

Perhaps one of the most popular examples of concurrent business process is order handling, when one of the steps - order packaging, follows provision materials that take place as two parallel processes, such as provision materials from existing stock and from external suppliers, and consists of invoicing and packaging materials that are used for this process, after which order can be shipped. If several concurrent processes carry out activities with the same data, another process may change the data, when the first process was interrupted. This may result in incorrect system result, which may not occur if the processes are executed serially, i.e. when the second process starts only when the first process is over.

The aim of the study initiated is to propose a methodology for the analysis of the business process(-es) in order to predict the probability of an incorrect concurrent execution of business process(-es). This would allow to configure transaction processing so that they cannot be processed incorrectly. It would point to potentially incorrect results whilst running business processes concurrently and allow system developers to redesign business process to prevent it. The aim of the paper is to present the preliminary version of the idea by explaining the rationale for the study, from basic concepts to the overview of the existing solutions related to the matter.

The paper deals with following issues: main concepts related to the topic presented and the rationale for the study (Section 2), review of existing solutions (Section 3), a description of the solution proposed (Section 4), conclusions and future work (Section 5).

II. Main Concepts and Rationale for the Study

A. Basic Concepts

Databases, both centralized and distributed, are often used to perform **transactions** - a set of data-dependent operations requested by the system user (some combinations of retrieval, update, deletion or insertion operations). The completion of the transaction is called a **commitment** and a cancellation before it is completed is called an **abort**.

One of the main properties of the transaction is an **atomicity**, which means that *all operations related to a particular transaction must be carried out or none of them can be performed*. That is, if a transaction is interrupted due to a failure, the transaction must be aborted so that its partial results are undone (i.e., rolled back) and, if the transaction is completed, the results are preserved (i.e., committed) despite subsequent failures [8]. It is also known that, in order to preserve data integrity, the DB system should provide a set of ACID properties, more precisely atomicity, consistency, isolation and durability. The study is closely linked to the **isolation** which is intended to hide the intermediate results of transaction from other concurrently executed transactions.

Concurrency control is a coordination of concurrent access to the DB while maintaining consistency of the data. Concurrency control techniques are divided into (1) **optimistic concurrency control**, which assumes that conflict is rare and it is possible to repair the potential losses caused by such violation, therefore, it delays the synchronization of transaction until transactions are close to their completion, and (2) **pessimistic concurrency control** that synchronizes the concurrent execution of transactions at the beginning of their execution life cycles, in other words, when an algorithm receives an operation, it makes a decision whether to accept, reject or delay this operation [9]. The pessimistic approach is considered safer than the optimistic, as it avoids potential problems rather than resolves them [10]. The optimistic concurrency control requires an effective repair mechanism and collisions cannot be destructive, so it is most effective when rarely conflicting writing operations take place [10]. Another type is **multiversion concurrency control**, according to which old versions of the data item are stored to increase concurrency. This algorithm has more flexibility in controlling the reads and writes order and it shows better performance in the cases with more "*read-only*" transactions, but needs a large storage space to preserve multiple versions of the data item.

B. Rationale for the Research

Practice demonstrates that there are cases when the data processing process $\mathbf{P_i}$ consists of several transactions $\mathbf{\{T_1, T_2, \ldots, T_n\}}$, where each transaction is independent and has its own **BEGIN TRANSACTION … COMMIT TRANSACTION** block. Thus, there are cases when within the one transaction ACID properties are fulfilled, but processes $\mathbf{P_1, P_2, \ldots, P_n}$ contain several steps where each contain one or more transaction calls and does not have a common BEGIN TRANSACTION … COMMIT TRANSACTION block. Therefore, the transaction management algorithms used by DBMS in this situation will not be able to prevent another process $\mathbf{P_m}$ from changing the data used by process $\mathbf{P_k}$ or reading and using intermediate results that may lead to incorrect results during the execution of the $\mathbf{P_k}$ process (between $\mathbf{T_i}$ and $\mathbf{T_{i+1}}$ calls). Such a case is possible either because of an error of programmers or because of the nature of the process, for instance, the process is so long that it is not reasonable to keep the total/ common resource locked during the process; therefore, there are **breakpoints** in the process where the shared resource is unlocked. It is even more difficult to analyze data processing processes, that are carried out within a number of systems, for which it is not even possible to create a common transaction.

This study addresses concurrent processes $\mathbf{P_a, P_b, \ldots, P_z}$ and their execution correctness which is defined according to the correctness of ACID. The DB is considered to be correct if it meets constraints imposed on it. The execution of a transaction brings a DB from one consistent state into another consistent state, thus, the execution of the transaction on the correct DB ensures its correctness after its execution. The result of serial execution of several transactions is correct, thus, if one process is carried out without concurrent execution of other process, the result is correct. If a series of several processes is performed, the result is correct. There may be several different, but correct results - depending on the order of execution, one of the possible correct results is achieved. As a result, an exact **criterion for any process and any input data correctness is the result obtained by one of the serial processes** (in line with [11]).

Concurrent execution is characterized by non-deterministic behavior that means that the repeated concurrent execution of several processes $\mathbf{P_i}$ with the same input data $\mathbf{X}$ yields different results that are caused by different sequence of synchronization events [12]. This fact makes concurrent processes difficult to test [13] since it is not enough to have a single sequence of processes that gives the correct result, it is vital to make sure that all possible orders provide correct results with all possible input data $\mathbf{X}$.

According to [12], one of the simplest and obvious approaches to deal with non-deterministic behavior is to execute it with a fixed input many times and hope that faults will be exposed by one of these executions (philosophy is close enough to optimistic concurrency) called **non-deterministic testing** and it is easy to carry out, but it can be very inefficient. It is possible that some behaviors of concurrent program are exercised many times while others are never exercised. An alternative approach is called **deterministic testing**, which forces a specified sequence of synchronization events to be exercised. This approach allows concurrent program to be tested with carefully selected synchronization sequences. The test sequences are usually selected from a static model of concurrent program or its design. However, accurate static models are often difficult to build for dynamic behaviors. An approach that combines non-deterministic and deterministic testing is **reachability testing**. This approach gets our preference in scope of this study as the most comprehensive one.

Another point is *how to ensure that all possible inputs will be tested*, since their number can be very high and even infinite. Here comes **symbolic execution** which operates on symbolic variables, where each possible initial state is considered instead of specific input data and all possible actions of the program are investigated [14]. In addition, [15] (and not only) have demonstrated that this technique significantly improves error finding in the way of resources. Thus, to cover all possible cases to be verified, the study launched will use symbolic execution.

III. STATE OF THE ART

Since a transaction is a concept closely linked to databases, let's first look at the more classic solutions in terms of databases, then focusing on more specific solutions.

Perhaps the most popular concept in terms of concurrent execution of multiple transactions is **isolation** - property that significantly simplify concurrent programming, since each transaction can be viewed separately, rather than having to consider all possible interleavings of their operations with other transactions. There are 4 "classical" isolation levels defined in ANSI SQL-92: *read uncommitted*, *read committed*, *repeatable read* and *[anomaly] serializable* [16]. In addition, one more isolation level appears to be popular and pre-defined in the number of DBMS, namely, *snapshot isolation*. However, depending on the isolation level, the number of possible phenomena can occur, more precisely, *dirty read*, *non-repeatable read*, *lost update*, and *phantom read*. In addition, isolation may also limit the applicability of transactions. For instance, according to [17], isolated transactions are incompatible with some common synchronization mechanisms, such as *barriers* and *ordinary condition variables*. More generally, while isolation seems to be an effective mechanism, it disallows programming idioms that require communication between transactions while they are active [17]. In addition, increased transaction isolation leads to a reduction in concurrency that is because isolation is usually done by blocking records, i.e. by setting locks, and if multiple rows are blocked, fewer transactions can be executed without temporary locking. While reduced concurrency is generally accepted as a compromise for higher transaction isolation level needed to maintain DB integrity, it may become a problem in interactive application with high reading/ writing activity [18].

The majority of the SQL-based solutions uses lock-based isolation levels, thereby let us briefly discuss the concept of locks and their diversity. **Locking** is a mechanism used to synchronize multiple users' simultaneous access to the same data by isolating so-called *critical code regions*. Before a transaction takes action, such as reading or modifying, it must protect itself from the consequences of another transaction that changes the same data. This can be achieved by requesting locks that can be of different mode, the most popular of which are *shared* and *exclusive*. In addition to these locking modes, there are three additional intention lock modes with multiple granularities, namely, *intension-shared* (IS), *intension-exclusive* (IX) and *shared and intension-exclusive* (SIX). The mode determines the extent to which the transaction depends on the data. No transaction can be assigned a lock that conflicts with a lock that was already assigned to these data for another transaction. It is usually managed by the concurrency control manager, which checks *what lock the data require to protect each resource* based on the setting of the access type and transaction isolation level, and *whether it is possible to assign this type of lock without violating the locks already assigned*. In the case of a SQL Server, if a transaction requests a lock that conflicts with a lock that was already assigned on the same data, a SQL Server Database Engine instance pauses a transaction that requests that lock until the first lock is released or removed. Modern DBMS provide a variety of granularity locks that allow to lock row (RID in the case of SQL Server), table (PAGE) and the whole DB (DATABASE). According to [9], the lock maintenance represents an overhead that is only needed if the conflicts occur; this overhead is justified only if the conflicts are rather likely (pessimistic assumption).

Another concept close enough and sometimes used with locks is **semaphore**. The aim of the semaphores is to order events, such as the execution of different critical regions. According to [17], two independent mechanisms (locks and semaphores) are not sufficient; rather, it would be better to have structured mechanisms that integrate them, such as **monitors** that can use the conditional variables. The authors argue that the current mechanisms do not provide a condition for synchronization with transactions, i.e. a mechanism that integrates transactions and ordering of events in a way that is analogous to the conditional variable. They conclude that the existing proposals for such synchronization include (a) conditional critical region (CCR) style transactions that allow the transaction to be executed only if a particular condition is fulfilled, and where the execution of the transaction is delayed until the condition is true, (b) a retrial construct that aborts the transaction that calls retry, and repeat its execution only when something in that transaction's read or write set is modified, and (c) the waiting construct or ordinary condition variable, which "punctuates", i.e., commits, the waiting transaction and begins a new transaction for the waiting thread on receipt of a notification from a concurrent transaction. But none of these proposals provides synchronous communication, such as *n-way rendezvous*, between concurrent transactions [17].

Therefore, there is a list of more specific solutions addressing this issue that were identified in the literature survey, founding several interesting studies that will be covered in this Section. The most popular and intuitive idea is the establishment of communication between transactions.

Considering that both approaches may have pros and cons, there are studies such as [9] that suggest **hybrid concurrency control** which is ensured by dynamically switching between a pessimistic and optimistic approach based on the value of conflict rate over the last *n* minutes, which should provide better performance. This algorithm uses adaptive resonance theory–based neural network when deciding whether to grant a lock or detecting a winner transaction. In addition, the parameters of this neural network are optimized with a modified gravitational search algorithm. The results of the developed algorithm application show that the algorithm proposed results in more than 35% reduction in the number of aborts in high-transaction rates compared with a strict two-phase locking.

One of the most impressive solutions is [8], proposing a distributed transaction approach, where all DB replicas are updated with a single, distributed transaction, which means that whenever a data item is updated with a transaction, all copies or replicas of that data item are updated as part of the same transaction. As a result, all replicas are completely synchronized. To ensure atomicity, the atomic commit protocol, such as **2 Phase Commit** (**2PC**) protocol, should be used in distributed transaction-based systems. The idea of 2PC

is to identify a unique decision for all replicas with respect to either committing or aborting a transaction and then executing that decision at all replicas. However, 2PC requires (1) each replicated database facility to submit a READY message before a transaction can be committed, which means that any site or link failure causes all activity to be halt until the site or link is repaired; (2) the transmission of at least 3 messages per replicated DB per transaction, that results in substantial communications resources and reduces the system's response time and throughput; (3) both, the coordinator and all participants must record the decision and the final outcome to stable storage, which involves 2 forced disk writes per participant per transaction, adding significant overhead. While some protocols have been proposed as a solution to the first problem, they impose even more communications overhead than 2PC. Thus, the author proposes to use a state machine approach to create a replicated, fault-tolerant DB system capable of coordinating the execution of concurrent transactions. In order to ensure the transaction atomicity and data consistency on each replicated DB server, application servers execute one of two new protocols – (1) **1 Phase Coordinated Commit** (**1PCC**) protocol which is more efficient by lacking the *rendezvous* step, but does not provide consistent serialization orders on all DB servers and thus only suitable for transactions whose results are independent on the order in which they are executed relative to other transactions, or (1) **2 Phase Coordinated Commit** (**2PCC**) protocol ensuring consistent serialization orders at each DB facility for all transactions which run at a serialization level guaranteeing repeatable reads and "phantom" protection. Both protocols are only concerned with the surviving cohorts - they can commit transactions despite the failure of the cohort. Consequently, the system provides improved fault-tolerance over traditional replicated systems, essentially ignoring failed replicas. This means that all surviving replicas are still processed because the failures are fully transparent to application clients.

[17] suggests using so-called **transaction communicator objects**. It is based on the *waiter* and the *notifier*. According to the proposal, this may extend transactional memory implementations to support transaction communicators and /or transaction condition variables for which transaction isolation is "relaxed" and through which concurrent transactions can be communicated and synchronized with each other. Transactional accesses to these objects must not be isolated unless they are called in communicator-isolating transactions. The waiter's transaction can invoke a wait method of a transaction condition variable, which can be added to a waiting list for the variable and be suspended while pending for a notification event from the notify method of the variable. The notifier's transaction may call a notify method of the variable, that may remove the waiter from the waiting list, schedule the waiter transaction for resumed execution and notify the waiter of the notification event. The waiter's transaction may only be committed if the corresponding notifier transaction commits. If the waiter's transaction is aborted, the notification may be forwarded to another waiter. This is also an example of the establishment of communication between transactions.

The presented idea significantly differs from the existing proposals. However, its idea can be [partly] compared with [19], in which authors proposed a debugging approach to Standard ML. It is not related to the study being launched, since the proposed study is intended to be applied to business processes, however, the authors (a) addressed the challenges of debugging a non-deterministic concurrent symbolic language, and (b) proposed an approach dealing with non-determinism. In addition, they also emphasize that the equivalence of behavior will be in the form of equality between possible execution histories.

IV. THE PROPOSED SOLUTION

The study proposes an algorithm for a business process that uses a transaction mechanism, analysis that aims to determine the probability of incorrect concurrent execution of multiple processes. The study is divided into two main parts: (1) **a modeling language** called **CPL-1** (Concurrent Process Language) that uses the transaction mechanism, and (2) an **algorithm** that, for every two processes defined in CPL-1, determines the probability of an incorrect execution of concurrent processes that is achieved in 3 interrelated steps: (1) creating a tree of possible scenarios, where each path represents one execution scenario; (2) defining the conditions for the feasibility of scenarios; (3) identifying incorrect concurrent execution conditions as a result of the solving conditional systems (Fig.1).

The proposed computing system is simple enough and it consists of (1) *processes*, (2) *transactions*, (3) *input data*, (4) *processor* which executes commands. As for *process*, (a) programs defined in CPL-1 can use local variables only, (b) variable can store a real number, (c) multiple variables can form a logical and numerical expressions, (d) numeric expressions can have only add- and subtraction operations, (e) programs can use operators assigning value to variable. The limitations regarding operations to be used are related to the fact that, if all arithmetic operations and complex functions are allowed to be applied, the conditions of scenarios may contain inequality systems that cannot be resolved. Thus, the nature of these operations is limited at this stage, to ensure systems to be solved are linear and easy to analyze, therefore, it is possible to find a solution, if any, or to prove that the solution does not exist.

For programs defined in CPL-1, (a) the process can call multiple *transactions*, which follows classical rules – transaction is executed entirely, either all operations defined in the transaction are executed, or they are cancelled, (b) transactions are not interrupted during their execution, and another process is not executed, (c) concurrent execution of multiple processes is performed before or after the transaction is completed - transaction ending command is a *breakpoint*, when switch to another process can take place.

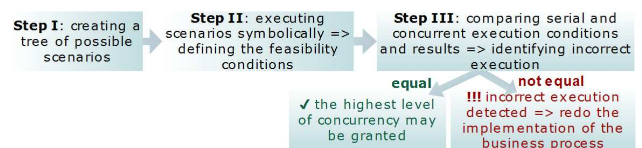

Fig. 1. Steps of the proposed algorithm.

For *input data*, which may be parameters or global resource, parameter values are passed on to a transaction when it is called, however, a global resource is a variable that is available for multiple transactions that can be executed concurrently and can read and write multiple transactions.

Therefore, CPL-1 allows such constructions as:

- START PROCESS … COMMIT PROCESS,
- BEGIN TRANSACTION … COMMIT TRANSACTION,
- READ(x, R),
- WRITE(x, R), which suppose read/ write of variable **x** value to the global resource **R**.

In addition, such logical constructions as "***IF L THEN BLOCK1 <ELSE BLOCK2> ENDIF***" are allowed, where block can contain one or more commands, for instance: $\mathbf{y} = \mathbf{EXPR}(\mathbf{x_1}, \mathbf{x_2}, .., \mathbf{x_n})$, where **EXPR** is linear expression, $\mathbf{x_1}, \mathbf{x_2}, .., \mathbf{x_n}$ are arguments and **y** – is a result.

Execution of one or more concurrent processes forms a *session*. The value of the global resource is defined at the beginning of the session. Processes are called by passing parameters to them. The process may contain multiple transaction calls. After each call and end of the transaction, there is possible breakpoint followed by a call of the next transaction and, possibly another process. CPL-1 does not suppose dealing with cycles. The programme contains only paths of finite length and the number of concurrent execution scenarios for multiple processes is also finite. This means that for each program defined in CPL-1, the finite **scenario tree** (Fig. 1 step 1) can be created where each scenario will be in the form of $\mathbf{P_1(T_1)>P_1(T_2(a,b))>P_2(T_1)>P_2(T_2(y,z))}$, where **a..z** are commands to be executed. The inability to allow cycles is due to the fact that it is not possible to create a complete test set (CTS) for programmes containing cycles and two-way counters.

CPL-1 formalized language is modeled before configuring the transaction process (step 0). Then, using the proposed analysis algorithm, the probability of incorrect process execution can be detected. If this possibility is revealed, the implementation of the business process must be redone, by reducing the risks identified. If such incorrect execution is not possible, the transaction mechanism may be granted the highest level of concurrency, since it has been demonstrated that the incorrect execution of the whole business process is not possible. For instance, if a billing operation is carried out, two possible mechanisms are possible, (1) with or (2) without reservation. The application of the proposed mechanism supports an intuitive assumption that a transactions execution without reservation can lead to an incorrect result, however, if reservation takes place, the results of all possible scenarios will be correct and the highest level of concurrency can be granted if such a mechanism is implemented (see our "toy example" in [20]). Let us take a look on how this can be ensured in more detail.

When the transactions are executed in a serial manner – each next transaction $\mathbf{T_{i+1}}$ begins when the previous $\mathbf{T_i}$ is ended - the result is not dependent on the time dimension. The concurrent execution of business processes that uses a transaction mechanism is affected by the order of the execution of multiple individual transactions. This results in two sets of process execution scenarios – (1) **concurrent (C)** - a set to be analyzed and (2) **serial (S)** - a set against which the first set to be analyzed.

The transaction is considered a "white box" in this study, so when the structure of the business process is known, there is hope to take advantage of the "white box" compared to the "black box". One of the most popular methods of "white box" analysis, namely the symbolic execution, allows the creation of execution conditions for any predefined scenario. Thus, symbolical execution of the corresponding commands, where real parameter variable $\mathbf{x_n}$ is replaced with symbolic value, results in the conditions for the feasibility of scenarios (Fig. 1 step 2). Such an approach is widely used in studies involving automatic test generation, such as Microsoft product IntelliTest, which is able to generate test data and a set of unit tests for the C# programs, performing an analysis of cases for each conditional path in the code, visiting each execution path. This approach allows creating tests with high code coverage (also in line with [21]). As a result, symbolic execution got a preference and is used to execute business process program that will result in setting up the conditions for execution scenarios. However, it is not a secret that symbolic execution can lead to a large number of case distinctions [14]. If two processes are interleaved, each step has two cases - either the first or the second process transition is executed. As a result, the size of the test tree is exponential in the number of transitions of the interleaved processes. However, it is very common that the order of executing interleaved transitions does not affect the resulting situation [14]. This fact will be the case for future studies.

When a feasibility tree (for the basics of the concept, see [22]) covering all possible scenarios for two business processes is created, it is necessary to identify cases where the results of concurrent and serial execution differ, which is done comparing the feasibility conditions and execution results of each concurrent execution scenario with the conditions and the results of each serial execution (Fig. 1 – step 3). If the feasibility conditions of the serial execution feasibility scenario are equal to one of the concurrent executions, but the results are different, an incorrect concurrent execution is detected, due to the fact that the result obtained cannot be achieved by performing any of the serial executions (Fig. 1 – step 3b). Analysis of different scenarios allows identifying inconsistencies between the results of execution of serial and concurrent scenarios. This indicates the probability of incorrect concurrent execution of the business processes (see an example in [20]).

Despite this time the case of two concurrent processes was mainly discussed, the concurrent execution of more than two processes is also possible. The proposed algorithm can find all possible concurrent scenarios and parameter value conditions for an arbitrary number of processes and transactions, leading to incorrect execution. However, depending on the number of processes, transactions, breakpoints, and the complexity of the programs, the size of scenario tree can grow rapidly. This means that tool to support concurrent execution analysis is required.

V. CONCLUSIONS

This preliminary paper deals with the concurrent execution of business processes. The main idea of the launched study is proposed, more precisely, to reveal the possibility that the business process is being implemented incorrectly by detecting the incorrect execution of concurrent processes. This, in turn, is linked to the complete test set (CTS) achieved through symbolic execution. The rationale for the study is defined according to existing studies, considering their pros and cons, and the key concepts that will be used to achieve a desirable result are provided.

The main outcome is the algorithm that determines for any two programs written in the proposed process description language CPL-1, whether incorrect concurrent execution is possible and, if possible, constructs a concurrent execution scenario, input data, and resource values that will lead to incorrect execution of processes. This makes it possible to determine whether it is possible to assign the highest level of concurrency, if incorrect execution is not possible, or if such a possibility is revealed, the implementation of the business process needs to be corrected by eliminating the risks identified (close to [23] idea). To sum up, analyzing business processes according to the procedure described allows to configure transaction processing optimally.

In the future, the proposed mechanism will be extended and implemented. An automatic solution is planned to be proposed to detect the possibility of incorrect result of concurrent execution of business processes, which should be easily applied to relatively simple but most classical business processes. This would allow the concurrent execution of processes that are certainly worth using without being afraid of the incorrect outcome. This also does not require resource-consuming activities to be carried out at the time of business process execution, that usually is solved by means of a mechanism establishing and managing communication between transactions that may fail in many cases.

ACKNOWLEDGMENT

The research leading to these results has received funding from the research project "Competence Centre of Information and Communication Technologies" of EU Structural funds, contract No. 1.2.1.1/18/A/003 signed between IT Competence Centre and Central Finance and Contracting Agency, Research No. 1.6 “Concurrence analysis in business process models".

REFERENCES

[1] J. A. Gohil, K. A. Popat, P. M. Dolia, “Comparative study and performance evaluation of JAG_TDB_CC concurrency control algorithm for temporal database”, In *2019 6th International Conference on Computing for Sustainable Global Development (INDIACom)*, 2019, pp. 548-553, IEEE.

[2] V. R. Kebande, I. Ray, “A generic digital forensic investigation framework for internet of things (IOT)”, In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 356-362, 2016, IEEE, doi:10.1109/FiCloud.2016.57

[3] V. R. Kebande, S. Malapane, N. M. Karie, H. S. Venter, R. D. Wario, “Towards an integrated digital forensic investigation framework for an IoT-based ecosystem”. In *2018 IEEE International Conference on Smart Internet of Things (SmartIoT)*, 2018, pp. 93-98, IEEE, DOI: 10.1109/SmartIoT.2018.00-19

[4] M. J. Islam, M. Mahin, A. Khatun, B. C. Debnath, S. Kabir, “digital forensic investigation framework for internet of things (IoT): A Comprehensive Approach”, In *2019 1st International Conference on Advances in Science, Engineering and Robotics Technology*, 2019, pp. 1-6, IEEE, DOI: 10.1109/ICASERT.2019.8934707

[5] Z. Wu, A. Abbas, X. Chen, S. U. J. Lee, “Classification of concurrent anomalies for iot software based support vector machine”, *Journal of Theoretical and Applied Information Technology*, 96(3), 2018.

[6] K. Hiraga, O. Tatebe, H. Kawashima, “PPMDS: A distributed metadata server based on nonblocking transactions”, In *2018 Fifth International Conference on Social Networks Analysis, Management and Security (SNAMS)*, 2018, pp. 202-208, IEEE, DOI: 10.1109/SNAMS.2018.8554478.

[7] G. Thakur, K. Sparks, A. Berres, V. Tansakul, S. Chinthavali, M. Whitehead ... E. Cranfill (2020). COVID-19 Joint Pandemic Modeling and Analysis Platform. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Modeling and Understanding the Spread of COVID-19* (pp. 43-52).

[8] R. K. Gostanian, J. E. Ahern, J. E., “Preforming concurrent transactions in a replicated database environment”, U.S. Patent No. 5,781,910, 1998,Washington, DC: U.S. Patent and Trademark Office.

[9] M. Sheikhan, S. Ahmadluei, “An intelligent hybrid optimistic/pessimistic concurrency control algorithm for centralized database systems using modified GSA-optimized ART neural model”, *Neural Computing and Applications*, pp. 1815-1829, 2013.

[10] M. Guzek, G. Danoy, P. Bouvry, “System design and implementation decisions for paramoise organisational model”, In *2013 Federated Conference on Computer Science and Information Systems*, 2013, pp. 999-1005, IEEE.

[11] D.Agrawal, A. El Abbadi, A. K. Singh (1993). Consistency and orderability: Semantics-based correctness criteria for databases. ACM Transactions on Database Systems (TODS), 18(3), 460-486.

[12] R. H. Carver, Y. Lei, “A general model for reachability testing of concurrent programs”. In *International Conference on Formal Engineering Methods*, 2004, pp. 76-98, Springer, Berlin, Heidelberg.

[13] S. M. Melo, J.C. Carver, P. S. Souza, S. R. Souza, “Empirical research on concurrent software testing: *A systematic mapping study”, Information and Software Technology*, 2019, pp. 226-251.

[14] M. Balser, “Verifying concurrent systems with symbolic execution: temporal reasoning is symbolic execution with a little induction”, 2006.

[15] N. Rungta, E. G. Mercer, W. Visser, “Efficient testing of concurrent programs with abstraction-guided symbolic execution”, In *International SPIN Workshop on Model Checking of Software*, 2009, pp. 174-191, Springer, Berlin, Heidelberg.

[16] S. Lütolf, S. ANSI SQL Isolation Levels. A Summary of the Original Paper “A Critique of ANSI. Version 2.1; https://wiki.hsr.ch/Datenbanken/files/Paper_ANSI_SQL_Isolation_Levels_Stefan_Luetolf_V2_1.pdf, 2014

[17] V. J. Marathe, V. M. Luchangco, “System and method for communication between concurrent transactions using transaction communicator objects”, U.S. Patent No 8,473,952, 2013.

[18] Microsoft SQL Docs. Concurrency Control (2017) available: https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/concurrency-control?view=sql-server-2017

[19] A. P. Tolmach, A. Appel, “Debuggable concurrency extensions for standard ML”. ACM SIGPLAN Notices, 1991, 26(12), pp. 120-131.

[20] J. Bicevskis, G. Karnitis, “Testing of execution of concurrent processes”, In *Proceedings of the 2020 conference on Databases and Information Systems XIV, DB&IS'2020* (in print).

[21] N. Tillmann, J. De Halleux, “Pex–white box test generation for. net. In International conference on tests and proofs”, 2008, pp. 134-153. *Springer, Berlin, Heidelberg*.

[22] S.E. Conry, R.A. Meyer, V.R. Lesser, “Multistage negotiation in distributed planning”, In *Readings in distributed artificial intelligence*, 1988, pp. 367-384. Morgan Kaufmann.

[23] K. Hiraga, O. Tatebe, H. Kawashima, “Scalable distributed metadata server based on nonblocking transactions”, *J. UCS*, pp. 89-106, 2020.