RC23948 (C0605-003) May 2, 2006
Computer Science

IBM Research Report

Guided Reasoning of Complex E-Business Process with Business Bug Patterns

Ke Xu[1,2], Ying Liu[1], Cheng Wu[2]

[1]IBM Research Division
China Research Laboratory
HaoHai Building, No. 7, 5th Street
ShangDi, Beijing 100085
China

[2]Automation Department
Tsinghua University
Beijing, China 100084

Guided Reasoning of Complex E-Business Process with Business Bug Patterns

Ke Xu [2] [1] (*xk02@mails.thu.edu.cn*), Ying Liu [2] (*aliceliu@cn.ibm.com*), Cheng Wu [1]
Automation Department, Tsinghua University, Beijing, China, 100084
IBM China Research Laboratory, Beijing, China, 100085

Abstract

With the growing complexity of e-business applications and the urgent need for ensuring its reliability and trustworthiness, much effort has been made to advocate the application of model checking in probing hidden flaws in these applications. This work devotes itself to the performance enhancement in reasoning e-business processes with model checking, which is a critical issue for making the reasoning of complex e-business applications more realistically applicable. Our major contribution lies in: (1) A set of business bug patterns are extracted from workflow patterns to exploit existing business knowledge in probing undesired violations in e-business processes; (2) The semantics of business bug patterns are formally captured with the IEEE standard of PSL; (3) Guided verification algorithms are development based on the above findings to enhance the performance of reasoning complex e-business applications. Their efficiencies are testified with three concrete business cases in banking and manufacturing domains with our business process verification toolkit of OPAL.

1. Introduction

As the Web is becoming a popular platform for implementing complex process centric e-business applications, e-business is facing with tremendous uncertainty due to its internal complexity and large scale interconnectivity. Urgent needs have arisen to assure highly secure and reliable e-business process development. Model Checking [1] has been shown to be a useful technique for probing potential bugs and finding hidden flaws in the e-business domain [2][3].

The basic idea of model checking is to search the state space of industrial application to witness its all violations of specific logical constraints defined by users. Therefore to make model checking more applicable to realistic large-scale models like e-business processes, its performance enhancement is a critical research direction. In this paper, we address this issue by introducing the idea of guided search of business bug patterns into model checking e-business processes. Though the idea of guided search itself is not new, the creativity and contribution of this work is concluded as:
1) A set of *Business Bug Patterns* are extracted from existing knowledge of workflow patterns as anti-patterns for e-business process enactment to detect its undesired behaviors and falsify its correctness.;
2) The semantics of these business bug patterns are formally captured with the IEEE standard of Property Specification Language (PSL) [4];
3) Consequently, corresponding guided search method is developed to enable a more efficient and less costly probing of business violations. The effectiveness of the method is proved by three different industrial business cases. Results show that our guided search gains radical performance enhancement in detecting business bugs especially for complex e-business applications.

2. Related Work

The basic idea of guided search in model checking is to constrain the traversal of system state space to the most "interesting" states so that system property can be proved more efficiently. In [5], 4 general heuristics are tested to speed up the probing of system bugs. [6] takes a next step by applying the empirical Bayes method to guide state space searching for explicit model checking. On the other hand, [7] and [8] focus on guided symbolic invariant checking and guided model checking of CTL respectively. Under-approximations and over-approximations of system models are obtained by "hints", the restriction on system behaviors, during the fix point iteration procedure. However, the *No Free Lunch Theorem* (NFLT) for optimization [9] has already taught us that "one cannot a priori expect an approach to yield good performance unless it explicitly specific system structures and knowledge". Hence, the difference between our work and the above ones is that we start from exploiting existing domain knowledge, i.e. the workflow pattern, for process-centric e-business applications. A set of anti-patterns called *business bug patterns* are identified and corresponding guided search algorithms are designed. Our guided search for model checking e-business process is implemented in the business process verification toolkit of OPAL [10], and its efficiency is tested in industrial cases with different sizes in hunting their potential pitfalls.

The organization of the paper is as follows. Section 3 investigates our identified Business Bug Patterns. In section 4, the guided verification of e-business process is implemented and tested in three different business cases. The verification results are also discussed. Conclusion of the paper can be found in section 5.

3. E-business Process and Its Bug Patterns

3.1. Preliminaries

In the process community, a similar approach as design patterns for object-oriented system development is taken to identify and codify the most useful and commonly occurred (control flow) relations in e-business processes. These relations are well categorized in the well known workflow patterns [11]. In this paper, the major concern of possible violations in e-business process is also focused on the control flow relations, i.e. the different ways that activities in an E-business process can be assembled to fulfill the application.

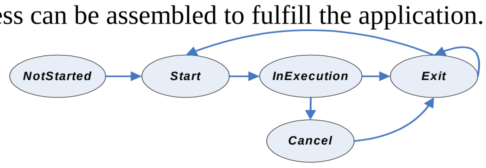

Figure 1. State transition for an activity

As extracted from [11] and [12] for capturing the semantics of workflow patterns, figure 1 illustrates the set of state transition relations for each activity in an e-business process. An activity is in its *Start* state if it is ready to be executed. The *execution* of an activity can be finished normally or possibly *Canceled* by some cancellation events. Either way, the activity will eventually remain being *Exited* until it is restarted again. Activities interact with each other with control structures like sequence, fork, join, decision, merge, etc. Each of these structures decides exactly how the change of state for one activity affects the state change of others. As a result, the enactment of an e-business process can be considered as all possible combinations of state transitions of its activities according to figure 1 (technical details of generating the state space of an e-business process can be found in our separate work [13]). Here it is reasonably assumed that each activity (and its instance) in a process can be uniquely identified by its own ID (denoted as uppercase letters: *A*, *B*, …).

While workflow pattern concludes the most common behaviors in a business process, its anti-pattern of *Business Bug Pattern* identifies the common behavioral violations in e-business processes. This is important since for complex e-business process, it is in practice more useful to probe its bugs (hence the name *Business Bug Pattern*) than prove its correctness [5]. It must be emphasized that business bug pattern is NOT used to model e-business processes and is independent of existing process modeling techniques. Rather, it specifies possible behavioral violations for defined e-business process models in order to reason about their reliabilities. In the rest of this section, the business bug patterns will be proposed according to the categories of workflow patterns. To have an intuitive and precise understanding of these bug patterns, their semantics are formally captured with the IEEE standard of PSL [4].

3.2. Basic Control Bug Pattern

3.2.1. Sequential Bug

SequentialBug(A, B) =
SimultaneousStart(A, B)∨NoResponse(A, B)

A simple *sequential* relation indicates that the execution of an activity *B* is always guarded by another activity *A*. To falsify this relation, it is focused on finding out (1) both *A* and *B* starts their execution simultaneously, or (2) *B* is never executed after *A* is exited. These two aspects are defined with two atomic bugs *SimultaneousStart(A, B)* and *NoResponse(A, B)* respectively, with their semantics captured in PSL [4] as:

SimultaneousStart(A, B)={[*];!A.Exit & B.Start}

*/*After some steps of execution, a state is reached in the process where B is started while the execution of A is not exited yet.*

NoResponse(A, B)= {[*];A.InExecution; A.Exit}
|->{B.InExecution[=0]}

*/*If A is finished in the process, no B will be executed afterwards*

Therefore, a *SequentialBug* holds when either *SimultaneousStart* or *NoResponse* is satisfied (as indicated with "∨"). Note that *SequentialBug* does not necessarily check whether *A* is possibly executed after *B* (because this is acceptable, e.g. *B* cycles back to *A* with the *Arbitrary Cycle* workflow pattern).

3.2.2. ParallelSplit Bug

ParallelSplitBug(A, {B, C})=
SequentialBug(A, B) ∨ SequentialBug(A, C)

The *ParallelSplit* pattern branches the execution of a process from an activity (*A*) to *Multiple* parallel paths (*B*, *C*). It is interesting to notice that the union of two *SequentialBug*s is sufficient to reverse the parallel split pattern, because in *ParallelSplit* two parallel activities do not necessarily imply that they are simultaneously executed. Hence the *ParallelSplitBug* pattern only specifies that *A*, *B* and *A*, *C* are both in sequential relation (i.e. *B*, *C* respond to the exit of *A* and will only be started after *A* exits), it does not constraint the execution order between *B* and *C*. This is the difference between this bug and *InterleavedParallelRoutingBug* as will be introduced in 3.6.2, which forbids the simultaneous execution of *B* and *C*.

3.2.3. Synchronization Bug

SynchronizationBug({B, C}, A) =
MultipleNoResponse({B, C}, A) ∨
SimultaneousStart(B, A)∨SimultaneousStart (C, A)

Contrary to *ParallelSplit*, *Synchronization* pattern converge *All* parallel paths *(B, C)* on a single activity (*A*) and continues the process. It orders that not only the execution of *B* and *C* be already exited when *A* starts (no *SimultaneousStart*), but also when *B*, *C* both exit their execution, *A* will be eventually started. Therefore a new atomic bug *MultipleNoResponse* is introduced to help capture the latter situation.

MultipleNoResponse({B, C, …, Z}, A)=
{[*];B.InExecution|C.InExecution|…|Z.InExecution; B.Exit &C.Exit &…& Z.Exit} |->{A.InExecution[=0]}

/ If B, C,…, Z are all exited in the process, no A will be eventually started afterwards*

As a result, the *SynchronizationBug* is the union of *MultipleNoResponse* and *SimultaneousStart*. Note that it does not check whether *B* and *C* can both be exited eventually in the process (because this is an acceptable case, e.g. when either of the activity is forced to be canceled by the *Cancellation* pattern in 3.7).

3.2.4. ExclusiveChoice Bug

ExclusiveChoiceBug(A, {B, C}) =
OverExecute (A, {B, C}) $\vee$
(NoResponse(A, B) $\wedge$ ***NoResponse(A, C))*** $\vee$
SimultaneousStart(A, B)$\vee$***SimultaneousStart (A, C)***

The *ExclusiveChoice* pattern branches the execution of a process from an activity (*A*) to *One* of several alternative paths (*B, C*). An *ExclusiveChoiceBug* contrarily indicates the situation in which either (1) both *B* and *C* starts after *A* exits (*OverExecute*); or (2) neither *B* nor *C* starts after *A* exits (*NoResponse*s); or (3) either *A, B* or *A, C* executes simultaneously. Therefore, the *ExclusiveChoiceBug* is constructed by the union of the above three aspects. Here "$\wedge$" indicates that both *NoResponse(A, B)* and *NoResponse(A, C)* need be satisfied to identify an *ExclusiveChoiceBug*. The semantics of the new atomic bug of *OverExecute* is:
OverExecute(A,{B,C})={[*];A.InExecution; A.Exit}
|-> *{{[*];B.InExecution } & {[*]; C.InExecution }}*
*/*If A is finished in the process, both B and C will be eventually executed afterwards*

3.2.5. SimpleMerge Bug

SimpleMergeBug({B, C}, A)=
PrematureStart({B, C}, A)$\vee$***InclusiveExit(B, C)*** $\vee$
(NoResponse(B, A) $\wedge$ ***NoResponse(C, A))***

The *SimpleMerge* pattern converges parallel paths (*B, C*) on a single activity (*A*) and continues the execution of the process. It is enough to falsify this pattern if one can find an execution path in the process on which (1) there is a state where *A* is in execution and neither *B* nor *C* exits; or (2) both *B* and *C* are exited; or (3) *B, A* and *C, A* both satisfy the *NoResponse* bug (because of the exclusive relation between the alternative paths). While the specification of the third situation is obvious, the first two are specified with the atomic bug of *PrematureStart* and *InclusiveExit* respectively.
PrematureStart ({B, C, …, Z}, A)= {[*]; A.Start &
(!B. Exit & !C. Exit & …& !Z. Exit)}
*/*When A is started in the process, none of its precedent activities B, C … and Z exit*
InclusiveExit (*B, C, …, Z)=*
{[*];B.Exit & C.Exit … & Z.Exit }
/ B, C and Z can all be exited in the process on a same state*

3.3. Advanced Branching & Synchronization Bug Pattern

3.3.1. MultiChoice Bug

MultiChoiceBug(A, {B,…,Z})=
$\bigwedge_{Act\in\{B,\dots,Z\}}$ ***NoResponse(A, Act)*** $\vee$
$\bigvee_{Act\in\{B,\dots,Z\}}$ ***SimultaneousStart (A, Act)***

Different from *ExclusiveChoice*, *MultipleChoice* pattern allows the choice of one or more branches (m-out-of-n for any m<=n) to execute based on the satisfaction of run-time conditions that are associated with each branch. Consequently, it is necessary to anticipate all possible scenarios for the different choice of branches in order to verify an e-business process against this behavior in all circumstances. Following this idea, the only indispensable scenario to falsify the multiple choice behavior is that none of the branch is ever chosen in the process model after the exit of the execution *A* ($\bigwedge_{Act\in\{B,\dots,Z\}}$*NoResponse(A, Act)*), or there exists an activity in the branch that simultaneously starts with *A* ($\bigvee_{Act\in\{B,\dots,Z\}}$ *SimultaneousStart (A, Act)*) .

3.3.2. SynchronizingMerge Bug

SynchronizingMergeBug({B,C,……,Z}, A)=
PrematureStart({B,C,……,Z}, A)

SynchronizationMerge joins branches that are spawned by a *MultipleChoice* Pattern. Similarly, it is also necessary to anticipate all possible scenarios for the successful execution of activities in different branches in order to find a *SynchronizingMergeBug*. Therefore, the only indispensable scenario in a *SynchronizingMergeBug* is that *A* already starts when none of the activities in branches is exited (*PrematureStart({B,C,D,……, Z}, A)*).

Note that in *SynchronizingMergeBug* it is not required that *A* must respond to the exit of any *Act*$\in$*{B,…,Z}* due to two reasons: (1) it is not possible to foresee which activity will definitely guard the start of *A*; (2) it is not possible to foresee which activity in the branches will definitely be exited eventually.

3.3.3. MultiMerge Bug

MultiMergeBug({B,C,……,Z}, A)=
SequentialBug(B, A)$\vee$ ***……***$\vee$***SequentialBug(Z, A)***

Similar to *SynchronizingMerge*, *MultipleMerge* pattern is also used to merge one or more enabled branches. However, while *SynchronizingMerge* waits for all enabled incoming branches to complete before continuing, in *MultipleMerge* each enabled incoming branch can independently trigger the start of the remaining e-business process. Therefore, the reverse of the above semantics is that there exists an *Act*$\in$*{B,…,Z}* such that *Act* and *A* follow the *SequentialBug* (either they start simultaneously or *A* does not respond to the finish of any *Act*). The result of the *MultiMergeBug* is the union of the *SequentialBug* for all *Act*$\in$*{B,…,Z}* in the branches and *A*.

3.3.4. ComplexJoin Bug

ComplexJoinBug({B,C,……,Z}, A, m, n)=
$\bigvee_{Act_i\in\{B,\dots,Z\}}$***PrematureStart(***$\{Act_1,\dots,Act_{n-m+1}\}$***, A)*** $\vee$
$\bigvee_{Act_i\in\{B,\dots,Z\}}$***InclusiveExit(***$Act_1,\dots,Act_{n+1}$***)*** $\vee$
$\bigwedge_{Act_i\in\{B,\dots,Z\}}$***MultipleNoResponse(***$\{Act_1,\dots,Act_n\}$***, A)***

The *ComplexJoin*, a.k.a the *M-out-of-N Join* (or its special case, the *Discriminator* pattern), lets through exactly the first *m* parallel branches at the convergence of *n* different branches. Any additional branch is blocked. The violation of *ComplexJoin* can be asserted under any of the following 3 situations: (1) *A* is started before enough (*m*) activities in the preceding branches are exited ($\vee_{Acti \in \{B,\ldots,Z\}}$ *PrematureStart*($\{Act_1,\ldots,Act_{n-m+1}\}$, *A*)); (2) More than *m* activities in different branches are exited ($\vee_{Acti \in \{B,\ldots,Z\}}$ *InclusiveExit* ($Act_1,\ldots,Act_{n+1}$)); (3) *A* does not respond to the *m* exited branches. The corresponding *ComplexJoinBug* is thus the union of the above three aspects.

3.4. Structural Bug Pattern

3.4.1. ArbitraryCycle Bug

ArbitraryCycleBug(A, B, C)=
SequentialBug(A, B) $\vee$ ***(NoResponse(B, A)*** $\wedge$
NoResponse(B, C))$\vee$***SimultaneousStart(B, C)***

The *ArbitraryCycle* just loops back (from *B*) to an activity (*A*), or continues the execution (from *B*) to another activity (*C*). It is simple to falsify it by asserting either (1) if *B* exits, neither C nor A actually starts; or (2) possible *SequentialBug* is satisfied between *A*, *B* or *B*, *C*. Note that here a *SimultaneousStart(B, C)* is used instead of *SequentialBug(B, C)* because the semantics of *NoResponse(B, C)* is already explicitly specified in *ArbitraryCycleBug*.

3.4.2. Implicit Termination Bug

No Related Bug Patterns Found.

Implicit Termination is the only pattern from which no corresponding bug patterns are found. This is because the intention of *Implicit Termination* is to help reduce the redundancy of the termination in a process by relaxing the restriction for a single global exit point. It essentially does not add expressiveness or impose any constraint on the behavior of an e-business process.

3.5. Multiple Instance Bug Pattern

3.5.1. WithoutSynchronization Bug

WithoutSynchronizationBug (b, A, B)=
RedundantInstance(b, A, B) $\vee$
SequentialBug(A, B)   **b is a Boolean condition**

The *WithoutSynchronization* allows performing multiple instances of an activity (*B*) after *A* with no overall synchronization of all the instances. As suggested in [11][12], it can be implemented as a parallel split in a loop. Denote the loop condition as *b*, *WithoutSynchronization* not only orders the sequential execution of *A* and *B*, but also iterates multiple executions of *B* when *b* is satisfied. Therefore, the *SequentialBug* is directly used to reverse the former semantics and a new atomic bug of *RedundantInstance* is introduced to falsify the latter.

RedundantInstance(b, A, B)= {[*];A.Exit & !b}
|->*{[*];B.Exit;B.Start}*

*/*When A is finished and b no longer holds, B still keeps to be re-executed*

3.5.2. WithDesignTimeKnowledge Bug

WithDesignTimeKnowledgeBug(A, B, n, C)=
ParallelSplitBug(A, {B_1, B_2, …, B_n})
$\vee$ ***SynchronizationBug({B_1, B_2, …, B_n}, C)***

The *WithDesignTimeKnowledge* performs a constant number (*n*) of instances of an activity (B_i) after *A* and synchronizes all these instances before the remaining e-business process (*C*) continues. This pattern can be simply realized by a *ParallelSplit* and *Synchronization* [12], with all activities between the two patterns to be the same. Therefore, a *WithDesignTimeKnowledgeBug* is thus defined by the union of the corresponding *ParallelSplitBug* and *SynchronizationBug*. Note that here the semantics of *ParallelSplitBug(A, {B1, B2, …, Bn}*) and *SynchronizationBug({B1, B2, …, Bn}, C)* are directly extended from 3.2.2 and 3.2.3 respectively:

ParallelSplitBug(A, {B1, B2, …, Bn})=
$\vee_{Act \in \{B1, B2, \ldots, Bn\}}$ *SequentialBug(A, Act)*

SynchronizationBug({B1, B2, …, Bn}, C) =
NoResponse({B1, B2, …, Bn}, C) $\vee$
$\vee_{Act \in \{B1, B2, \ldots, Bn\}}$ *SimultaneousExecution (Act, C)*

3.5.3. WithRunTimeKnowledge Bug

WithRunTimeKnowledgeBug(A, B, n, C)=
MultiChoiceBug(A, {B_1, B_2, …, B_n}) $\vee$
SynchronizingMergeBug({B_1, B_2, …, B_n}, A)

The *WithRunTimeKnowledge* specifies a similar behavior with 3.5.2, except that the actual number of activity instances is known at runtime. Therefore, the difference between *WithRunTimeKnowledgeBug* and *WithDesignTimeKnowledgeBug* is that the *MultiChoiceBug* and *SynchronizingMergeBug* are used to replace the *ParallelSplitBug* and *SynchronizationBug* respectively. The purpose of the replacement is to consider the case for any possible number of activity instances while it is still unknown due to the runtime dependent nature of this pattern. Note that a maximum allowed instance number (*n*) is needed here for *WithRunTimeKnowledgeBug* in order to avoid the state space of the e-business process to be infinite.

3.5.4. WithoutRunTimeKnowledge Bug

WithoutRunTimeKnowledgeBug(A, B, b, n, C)=
WithRunTimeKnowledgeBug(A, B, n, C) $\vee$
RedundantInstance(b, A, B)

The *WithoutRunTimeKnowledge* further generalizes 3.5.3 by leaving the required number of activity instances undetermined as late as possible until some evaluation point during the actual processing of the activity. As suggested in [12], its implementation is similar to 3.5.3 expect that the governing loop is not a for loop implying the required number of instances but a while loop with the evaluation of condition *b* for the iteration of activity instances. As a result, the definition of *WithoutRunTimeKnowledgeBug* not only checks the existence of *WithRunTimeKnowledgeBug*, but additionally falsifies whether *B* can be re-instantiated after the evaluation of *b* is failed by the *RedundantInstance* bug defined in 3.5.1.

3.6. State-Based Bug Pattern

3.6.1. DeferredChoice Bug

DeferredChoiceBug(A, {b, B, c, C})=
OverExecute(A, {B, C}) $\veebar$
(NoResponseOnEvent(A, b, B)
$\wedge$ ***NoResponseOnEvent(A, c, C))*** $\vee$
SimultaneousStart(A, B)$\vee$***SimultaneousStart(A, C)***
b, c are Boolean conditions

A *DeferredChoice* is much like an *ExclusiveChoice* except that the branch to be taken to execute is not chosen immediately but is instead deferred until an event (*b*, *c* for activity B, C respectively) occurs. To implement a *DeferredChoiceBug*, revisions should be made based on the *ExclusiveChoiceBug*. A new *NoResponseOnEvent* bug is introduced instead of the simple *NoResponse* bug to fulfill the semantics of waiting external events in *DeferredChoiceBug*:
NoResponseOnEvent(A, b, B)=
{[*];A.InExecution; A.Exit}|->
{B.InExecution[=0]} & {[*]; b}
*/*After A is (possibly) finished in the process, event b occurs but no B is ever in execution*

3.6.2. InterleavedParallelRouting Bug

InterleavedParallelRoutingBug (A, {B, C})=
SimultaneousStart(B, C)$\vee$***SimultaneousStart(C, B)***
$\vee$***SequentialBug(A, B)*** $\vee$ ***SequentialBug(A, C)***

The intention of an *InterleavedParallelRouting* is to perform several activities (*B*, C) in *arbitrary* sequential orders after *A*. The natural implementation of the *InterleavedParallelRoutingBug* is shown above. Two *SequentialBugs* are used to make sure whether *B* or *C* will be eventually executed after *A* exits and whether they are simultaneously started with *A*. Meanwhile, *SimultaneousStart* is also used to identify whether the execution of B and C is against the interleaving mode.

3.6.3. Milestone Bug

MilestoneBug (en, dis, A)= PrematureStart(en, A)
$\vee$ ***RedundantInstance(dis, A, A)***
en, dis are Boolean conditions

The Milestone pattern describes the scenario that an activity (*A*) can be executed after an enabling event (*en*) occurs and before the occurrence of a disabling event (*dis*). The definition of *MilestoneBug* is straightforward, with *PrematureStart* falsifying "enabling" semantics and *RedundantInstance* falsifying the "disabling" semantics. Note here *PrematureStart(en, A)* is defined as: *PrematureStart(en, A)= {[*]; A.Start &(!en)}*.

3.7. Cancellation Bug Pattern

3.7.1. CancelActivity Bug

CancelActivityBug(cancel, A)
cancel is a Boolean condition

In *CancelActivity*, an activity (*A*) is cancelled on a specific cancellation trigger (*cancel*). Since no previous bug patterns can be used directly to falsify this behavior, a *CancelActivityBug* is independently introduced below:
CancelActivityBug(cancel, A)= {[*];cancel&
(A.InExecution| A.Start)}|->{A.Cancel[=0]}
The above definition asserts that when *A* is started or in its execution and the cancellation trigger (*cancel*) arrives, *A* is never cancelled afterwards.

3.7.2. CancelCase Bug

CancelCaseBug(cancel, Process)=
$\bigvee_{Act\in Process}$ ***CancelActivityBug (cancel, Act)***
cancel is a Boolean condition

CancelCase specifies the behavior of stopping the execution of the entire process on a specific cancellation trigger. Therefore a *CancelCaseBug* holds if any of the activity in the process satisfies the *CancelActivityBug*. It means when a cancellation trigger is arrived, there exists an activity within the range of *Process* which is started or in execution but will not be canceled afterwards. Note that the behavior of the activities after cancellation (when and how they will be reinitiated) is left unspecified in *CancelCaseBug*.

4. Guided Search of Business Bug Patterns

4.1. Guided Search of Interesting States

The previously concluded business bug pattern enjoys two characteristics: (1) all of them are evaluated on a single execution path in the e-business process model (no universal qualifier is needed); (2) the specifications of these bug patterns involves only the evaluation on paths where time advances monotonically. Besides, the business bug patterns provide us the conditions for target states which can be used to assert the discovery of the corresponding business bug in e-business processes (e.g. *!A.Exit & B.Start* in *SimultaneousStart*). We call these states the *Commitment States*. Based on these features, it is enlightened to develop a guided search mechanism to enhance the efficiency of model checking e-business process models for common behavioral violations.

The idea of guided search is to always find and follow "interesting states" during the traversal of model state space to quickly detect the existence of a business bug in e-business processes. Given a commitment state *CS*, an ***interesting state*** in a state set *SS* is defined as the state that can transit to *CS* within the least steps. More specifically, denote:
M(m): the complete state space (universe) of an e-business process *m*, with its initial state on which all of the activities are *NotStarted*;
$\boldsymbol{S(m)=\{s(act_1),s(act_2),\ldots\}}$: A state in *M(m)* which is encoded as the states of all activities in *m*, where $act_i \in m$ & $s(act_i) \in \{act_i.NotStart,\ act_i.Start,\ act_i.InExecution,\ act_i.Cancel,\ act_i.Exit\}$;

Thus the distance between *two states on a same activity* $\boldsymbol{D(s(act)_1, s(act)_2)}$ is defined as the least number of transitions in figure 1 that $s(act)_1$ can get to $s(act)_2$.

E.g. ***D(act.Start, act.Exit)=2*** (in the case when a simple Boolean value is considered (e.g. *en, dis*), the result is either 0 or 1). Therefore, the distance between *two states in process m* is defined as the average of *D*:

$$\boldsymbol{D\_S(S(m)_1, S(m)_2)} = \sum_i D(s(act)_{1i}, s(act)_{2i}) \,/\, |\,S(m)|$$

The ***interesting states*** for a given commitment state *CS* in state set *SS* are thus:

$$\boldsymbol{S(m)\_CS} = \{S(m) | S(m) \in SS,\ \forall S'(m) \in SS, \text{ there holds: } D\_S(S(m), CS) < D\_S(S'(m), CS)\}$$

The definition implies that interesting states to a commitment state *CS* always hold the *shortest distance* to *CS*. Naturally, the interesting level of states for *CS* in state set *SS* can also be defined. Define the interesting level of $S(m)\_CS1 = S(m)\_CS$ is 1 (write as $Lv(S(m)\_CS_1)=1$), then $S(m)\_CS_n$ is the level *n* interesting states in *SS* ***iff*** $S(m)\_CS_n$ is the level 1 interesting states in $\{SS - S(m)\_CS_1 - \ldots - S(m)\_CS_{n-1}\}$.

Practically, multiple commitment states are often considered when evaluating business bug patterns. E.g., for *SimultaneousStart* bug (3.2.1), there can be multiple states satisfying *!A.Exit & B.Start*. Arriving each state may witness its occurrence. For *OverExecute* in 3.2.4, on the other hand, commitment states for *B.InExecution* and *C.InExecution* should both be reached to witness its satisfaction. Therefore, it is further defined:

$S(m)\_CS' = \min\{S(m)\_CS_1, S(m)\_CS_2\}$, where *CS' implies* ***either*** CS_1 ***or*** CS_2; and

$S(m)\_CS' = \max\{S(m)\_CS_1, S(m)\_CS_2\}$, where *CS' implies* ***both*** CS_1 ***and*** CS_2.

With the above definitions, a guided model checking algorithm for e-business processes is implemented by further improving the breadth-first symbolic model checking in [1]. Two major revisions are made in the newly implemented algorithm of *GuidedBugHunting* (GBH) and *GuidedBugHuntingBackTrace* (GBH-BT), as illustrated in figure 2: (1) A forward iteration of process traversal is implemented instead; (2) Additional distance information is used to define interesting states to guide the search of the process. (*Pre*)*Image(P)* is used to indicate the functions for computing formulas of (pre)images (i.e. predecessors / successors of a state).

The essence of GBH and GBH-BT is *CompInteresting(fP, CSs, Lv)*, which computes the formula representation for interesting states in the image of current states (*fP*) from target commitment states *CS*s under interesting level *Lv*. Consequently the algorithms always stick to the interesting states first, and thus reduce the size of states evaluated in each step of state traversal and the time for bug hunting. The difference between GBH and GBH-BT is that GBH always (and only) considers the interesting states during verification. Therefore, it does not traverse the complete state space of an e-business process and probes its violations on its *under-approximation*. GBH-BT, on the other hand, traces back to uninteresting states and uses them as the initial points for re-evaluating process models when bugs are not found on interesting states. Note both two algorithms are self-adaptive, in that when no interesting states are found (*fP=Guided*), it will restrict the evaluation condition by either reduce the interesting level or renew the commitment states by computing their preimages.

The computation complexity of *CompInteresting* is *O(m*n)*, where m is the size of the current traversing states (*fP*) and *n* is the size of the considered commitment states. While the size of *n* remains small for each iteration of the e-business process, *m* is also a small number due to two reasons: (1) microscopically *m* is restricted by traversing only the interesting states; (2) macroscopically, different from hardware or software designs, the state space for E-business processes is often large in its depth and small in its breadth.

4.2. Application of Guided Bug Searching

As shown in figure 3 and 4, three business scenarios of different complexities, an abstract open account e-business process for banking system (*SimpleBancking*), its detailed implementation (*ComplexBanking*) and an e-business process for sofa-manufacturing (*SofaManuf*),

```
Init: the formula representation for initial set of states;
CSs: the formula representation for commitment states;
Gate: the maximum number for computing preimages;
MLv: the maximum interesting level;
Procedure GuidedBugHunting (GBH)
1: Define P=Init=Reached, pre=1, Lv=MLv
2: Define fP=Image(P)
3: Define fI=CompInteresting(fP, CSs, Lv)
4: Define Guided= $fP \wedge fI$
5: If Witness(Guided)
6:        Report CS found, return Guided
7: If fP= Guided
8:        If Lv>1    Lv--, goto 3;
9:        If pre<Gate
10:           pre++;  CSs=PreImage(CSs), goto 3
11: Define Union=$Guided \vee Reached$
12: If Union=Reached
          Report Bug not found, return false
13: P= $Guided \wedge \neg Reached$
14: Reached=Union, goto 2
```

```
Procedure GuidedBugHuntingBackTrace (GBH-BT)
1: Define P=Init=Reached, pre=1, Lv=MLv
2: clean stack to make it empty
3: Define fP=Image(P)
4: Define fI=ComInteresting(fP, CSs, Lv)
5: Define Guided= $fP \wedge fI$
6: If Witness(Guided)
7:        Report CS found, return Guided
8: If fP= Guided
9:        If Lv>1  Lv--, goto 3
10:       If pre<Gate
11:           pre++;  CSs=PreImage(CSs), goto 4
12: Define UnInterested= $fP \wedge \neg fI$
13: If UnInterested!=false
14:       Insert UnInterested into stack
15: Define Union=$Guided \vee Reached$
16: If Union=Reached
17:       If stack is not empty
18:           P= remove Head of stack
19:           Union=Reached=$P \vee Reached$;  goto 3
20:       Else Report Bug not found, return false
21: P= $Guided \wedge \neg Reached$
22: Reached=Union, goto 3
```

Figure 2. Algorithms for Guided Search

are used as test cases. The models are built with IBM's Websphere Business Integrator (WBI). Detailed introduction of the above process can also be referred in [10]. Our OPAL[10] (Open Process AnaLyzer) toolkit is the implementation environment for showing the efficiency of our approach. OPAL is a model checking based verifier tuned for business processes. It accepts WBI process models as input and automatically records and manages the state of the process by monitoring the transition of corresponding activities. Details of the formalization and verification methodology of OPAL can be found in [10]. To prove the effectiveness of our proposed approach, both the original *Breadth-First Symbolic Model Checking* (BF-SMC) of CTL[1] and our *GuidedBugHunting* (GBH and GBH-BT) are implemented in OPAL. Tables 1-3 illustrate the results (*Gate=1*, *MLv=2*) for different business bug patterns, which are run on the PC of 1.73GHz and 2.5G RAM.

4.3. Result Discussion

The above choice of business scenarios and business bug patterns shows the diversity in their complexity (reachable states varies from 10^3 to 10^6). For *SimpleBancking*, GBH does not show much advantage over BF-SMC, the performance of GBH-BT is even worse than BF-SMC. This is because the scale of this e-business process is so small that makes the additional computation of interesting states comparatively costly. However, the small application is surely not our focus. The advantage of our approaches becomes more obvious for latter two complex applications. For models where bugs do exist (Result=*false*), GBH-BT can always detect these bugs faster than BF-SMC since it always focuses on the most promising paths on which bugs can be found. However, when specific bugs do not exist in the model (e.g. B3 and B9 in table 1-3), GBH-BT terminates slower than BF-SMC since GBH-BT not only traverses the whole model state space, but also does additional computation for interesting states and stack operations. On the other hand, GBH out-performs BF-SMC in both cases in the application. This is because GBH follows only the interesting paths that may lead to a bug and wastes no time on the un-interesting ones. However, the drawback of GBH is that it only evaluates a bug on an *under-approximation* of the target application. Consequently, GBH can be used to falsify an e-business process with specific bugs, but it cannot fully prove the correctness of the process even if the approach returns the result of *true*.

As a matter of fact, GBH can be regarded as a special case for GBH-BT with the stack size set to be 0 (line 14 in GBH-BT). Therefore, three conclusions can be drawn based on the above results: (1)Both GBH and GBH-BT can be used to effectively detect behavioral violations in complex e-business processes by guiding the search to interesting states first; (2)GBH is most efficient for complex e-business process where the application is too large for model checking to run to completion; (3) The verification performance and completeness of state space traversal can be balanced by setting the allowed stack size in GBH-BT (line 14) according to the complexity of target e-business process.

5. Conclusion

In accordance to existing workflow patterns, a set of

Reachable States			958 (2^9.904)		
Total States			8.941*10^11 (2^39.702)		
BF-SMC		**GBH**		**GBH-BT**	
Result	**Time**	**Result**	**Time**	**Result**	**Time**
***B1**: SequentialBug(IdentifyCustomer, OpenAccount)*					
false	0.140	*false*	0.156	*false*	0.189
***B2**:ParallelSplitBug(RetainCustomerInfo, {SelectService, PrepareOpening })*					
false	0.116	*false*	0.078	*false*	0.136
***B3**: SynchronizingMergeBug({RetrieveCustomerDetail, SelectService, PrepareOpen}, OpenAccount)*					
true	0.265	*true*	0.141	*true*	0.357
***B4**: WithDesignTimeKnowledgeBug (RetrieveCustomerInfo, Review, IdentifyCustomer, 2)*					
false	0.359	*false*	0.384	*false*	0.429
***B5**: MilestoneBug(OpenAccount.Finish, ActivateAccount.PreStart, ValidateAccount)*					
false	0.209	*false*	0.186	*false*	0.215

Table 1. Results for *SimpleBancking* (in seconds)

Reachable States			82612 (2^16.334)		
Total States			4.213*10^13 (2^45.260)		
BF-SMC		**GBH**		**GBH-BT**	
Result	**Time**	**Result**	**Time**	**Result**	**Time**
***B1**: SequentialBug(IdentifyCustomer, OpenAccount)*					
false	82.678	*false*	26.624	*false*	36.233
***B2**:ParallelSplitBug(RetainCustomerInfo, {SelectService, PrepareOpening })*					
false	72.656	*false*	28.187	*false*	38.078
***B3**: SynchronizingMergeBug({RetrieveCustomerDetail, SelectService, PrepareOpen}, OpenAccount)*					
true	59.016	*true*	10.152	*true*	72.598
***B4**: WithDesignTimeKnowledgeBug(RetrieveCustomerInfo, Review, IdentifyCustomer, 2)*					
false	164.92	*false*	27.422	*false*	40.506
***B5**: MilestoneBug(OpenAccount.Finish, ActivateAccount.PreStart, ValidateAccount)*					
false	90.360	*false*	22.671	*false*	29.219

Table 2. Results for *ComplexBanking* (in seconds)

Reachable States			193524 (2^17.5622)		
Total States			1.133*10^14 (2^46.69)		
BF-SMC		**GBH**		**GBH-BT**	
Result	**Time**	**Result**	**Time**	**Result**	**Time**
***B6**: SequentialBug(setslipover, fixchairshape)*					
false	143.08	*false*	16.422	*false*	23.578
***B7**: ExclusiveChoiceBug(SendProductionOrder, {GetSewingMaterial, GetSamplingMaterial}) & (SequentialBug(GetSamplingMaterial, Sampling) | InterleavedParallelRoutingBug(GetSewingMaterial, CutSponge, SewLeather))*					
false	401.38	*false*	63.488	*false*	268.99
***B8**: WithoutSynchronizationBug(BindSponge, FinishNailing.NotStarted, setslipover)*					
true	465.35	*true*	27.891	*true*	303.92
***B9**: CancelActivityBug(ReportOOS.PreStart, Sampling) | CancelActivityBug(ReportOOS. PreStart, CutSponge)*					
true	294.50	*true*	42.281	*true*	332.30

Table 3. Results for *SofaManuf* (in seconds)

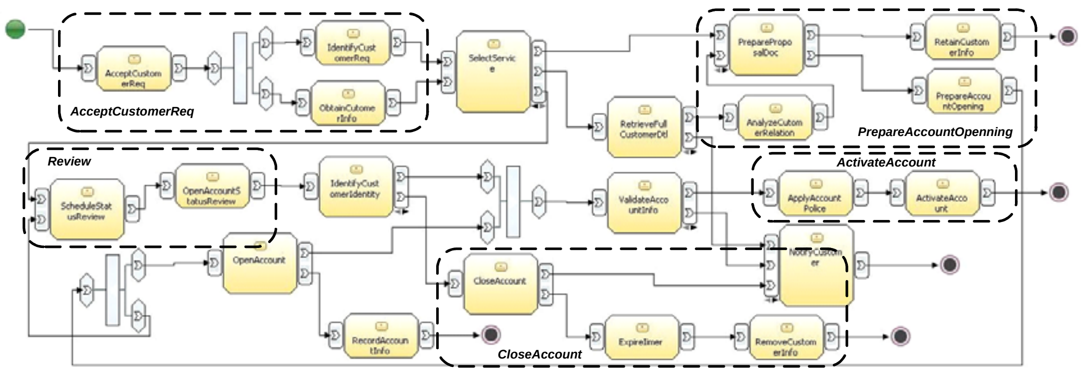

Figure 3. Sample Banking E-business Application (detailed behaviors in dotted rectangles can be abstracted)

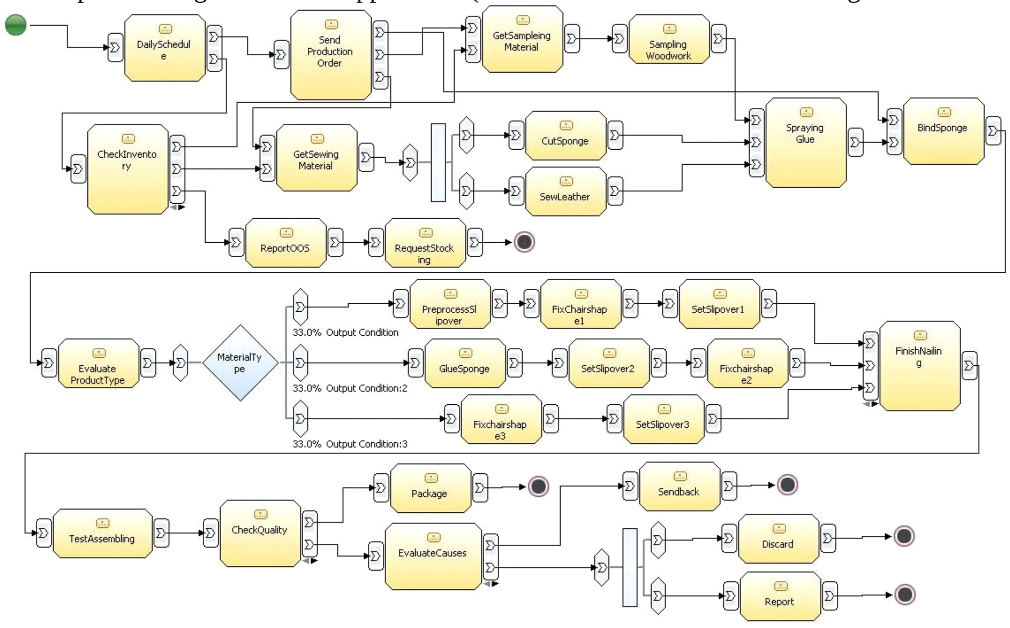

Figure 4. Sample Manufacturing E-business Application

business bug patterns are identified in this paper to capture the common behavioral violations in complex e-business processes. The precise semantics of the bug patterns are formally captured with IEEE standard of PSL. Guided searching for these bug patterns is also proposed to enhance the efficiency of reasoning e-business applications to ensure their trustworthiness. The approach is implemented in our business process verification toolkit of OPAL and testified on three different business scenarios in both banking and sofa-manufacturing domain. Results show that our approach is especially useful for large e-business applications compared to traditional model checking approach.

As previously concluded, our future work involves the careful study of the relation between stack size in GBH-BT and verification performance to make it a more systematic approach.

6. References

[1] E. Clarke, A. Biere. “Bounded Model Checking Using Satisfiability Solving”. Formal Methods in System Design, 19, 2001, pp. 7–34.

[2] W. L. Wang, Z. Hidvegi. ”E-process design and assurance using model checking”, Computer, 33, 10, pp. 48-53.

[3] B. Anderson, J. Hansen, et al. ”Model checking for design and assurance of e-Business processes”, Decision Support Systems, 39, 3, pp. 333-344.

[4] D. Geist. “The PSL/Sugar specification language a language for all seasons”. Lecture Notes in Computer Science, 2800, 2003, pp. 3-3.

[5] C. H. Yang, L. D. David, “Validation with guided search of the state space”. Proc. Annual ACM IEEE Design Automation Conference, 1998, pp. 599-604.

[6] K. Seppi, M. Jones, et al, “Guided model checking with a bayesian meta-heuristic”. Proc. 4th Int. Conf. on Application of Concurrency to System Design, 2004, pp. 217-226.

[7] K. Ravi1, F. Somenzi, “Hints to accelerate symbolic traversal”, Lecture Notes in Computer Science, 1703, 1999, pp. 250-266

[8] R. Bloem, K. Ravi, et al. ”Symbolic Guided Search for CTL model checking”. Proc. Annual ACM IEEE Design Automation Conference, 2000, pp. 29-34.

[9] Y. C. Ho, D. L. Pepyne, “Simple explanation of the No Free Lunch Theorem of Optimization”. Proc. IEEE Conf. on Decision and Control, v5, 2001, pp. 4409-4414.

[10] K. Xu, Y. Liu, C. Wu. “BPSL modeler - visual notation language for intuitive business property reasoning” Electronic Notes in Theoretical Computer Science, 2006, pp. 205-215.

[11] van der Aalst W.M.P, ter Hofstede, A.H.M. et al, “Work-flow patterns”. Distributed and Parallel Databases, 14, 2003, pp. 5－51.

[12] Michael H. *Essential Business Process Modeling*, O’Reilly Press, 2005.

[13] K Xu, Y. X. Wang., C Wu. “Ensuring secure and robust grid applications – From a formal method point of view”, Lecture Notes in Computer Science, 3947, 2006, pp. 537-546