Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 295 (2013) 79–93

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Model Slicing Method for Workflow Verification

Fazle Rabbi Hao Wang Wendy MacCaull Adrian Rutle

Centre for Logic and Information
St. Francis Xavier University, Canada
{rfazle, hwang, wmaccaul, arutle}@stfx.ca

Abstract

Workflow systems increase productivity and quality of service; however, defects in a workflow model may have severe consequences. While model checking techniques can be used to verify the correctness of a workflow model, these techniques typically suffer from the *state explosion* problem. We propose a model slicing algorithm with a formal proof to address this problem. The algorithm is integrated into our NOVA Workflow framework, which facilitates design, verification, execution, and error-handling. An experimental result has been presented to show that the proposed algorithm makes the verification more efficient in terms of state space and hence for memory and time usage.

Keywords: Model slicing, formal verification, workflow modelling, model checking, linear temporal logic

1 Introduction

Workflows coordinate activities performed by various participants, e.g., persons, robots or software components, in order to achieve a business goal. Today workflow models are frequently used to describe the behaviour of software components in distributed and heterogeneous environments with characteristics like concurrency, resource sharing, and synchronisation. In such a collaborative environment, workflow systems could be used for controlling runtime execution and service orchestration, based on a workflow model describing the behaviour of the components.

Many of today's workflows are complex, requiring a high degree of flexibility, massive data and knowledge management. However, the resulting implementations of unverified, complex workflow models are at risk of undesirable runtime executions. Model checking or other similar verification techniques are required to ensure that these process models exhibit the desired behaviour. The time complexity of model checking algorithms depends on the size of the transition system which is exponential in the number of variables, concurrent components, and channels. This problem is commonly known as the *state explosion problem* [4]. While current model

http://dx.doi.org/10.1016/j.entcs.2013.04.007

checkers are much more powerful than their predecessors, they still frequently suffer from this problem. Much research has been done in this field during the past decade. While there are many techniques to optimize model checking algorithms, e.g., *Partial Order Reduction* (POR) [15], *Symmetry-based Reduction* [9], the category of techniques called *Model Abstraction* [4] is also very important to alleviate the state explosion problem. These techniques abstract away irrelevant details w.r.t. the properties being verified before the system model is input to the model checker.

There are two types of model abstraction techniques: *data abstraction*, which uses a smaller set of data values to represent the actual values of the system, and *model slicing*, which eliminates model components that will not affect the truth value of the property being verified. In this paper we focus on the latter technique.

While the problem of *software program slicing* [7] is a well-studied topic, there are relatively few works in slicing for formal diagrammatic languages, even fewer for workflow modelling languages. This paper develops and integrates model abstraction techniques into a workflow framework.

Our group developed NOVA Workflow, a framework for workflow design, verification, execution, and error-handling, which has three components: an editor, an engine and a translator. Using the NOVA Editor one can graphically model workflows for many application domains, e.g., healthcare protocols, business processes, scientific workflows, etc., using the *Compensable Workflow Modeling Language* (CWML) [17] and write business logic for the tasks. These workflow models are executed by the NOVA Engine. Before execution, the workflow models are automatically translated to DVE, the input language of the DiVinE [3] model checker, using the NOVA Translator [17]. Although we could model a large workflow and translate it into a model checking program, the time and space required for the verification was sometimes unacceptable. Our experiments showed that even though DiVinE is equipped with POR and many other heuristics, it requires a huge amount of memory and time for the verification of a large model.

In this paper we present a model slicing algorithm for the workflows constructed using CWML. The algorithm has been implemented in the NOVA Translator. It takes a workflow model and the specification of a temporal logic formula ϕ and reduces the model in such a way that the truth of ϕ is preserved and reflected. Linear Temporal Logic (LTL) is a powerful tool for reasoning about system properties that vary over time. LTL is a type of temporal logic which, in addition to the classical propositional logical operators, uses the temporal operators: *always* (G), *eventually* (F), *until* (U), and *next time* (X). Its full semantics may be found in [4]. The slicing algorithm presented in this paper works with LTL_{-X} formulae (the subset of LTL formulae not containing the X operator [4]).

We give the details of the proof of the equivalence of the original model and reduced model generated by our algorithm. Moreover, we show how effectively the proposed method reduces the size of the state space. We expect the proposed algorithm to be easily applied to any block-structured modelling language, e.g., BPEL [8]. Note that the algorithm deals with the feature of data-awareness now commonly found in workflow modelling languages. We show the applicability and

effectiveness of the method with a fairly big model of a healthcare workflow.

Fig. 1 shows an overview over the different steps in the proposed approach of the paper. Our workflow models M are specified by CWML. Each model M is serialised as a corresponding textual expression valid w.r.t. the BNFs in Definitions 2.2 and 2.3. We use the Text2TST algorithm (see Algorithm 1) to parse the expression and generate a Task Syntax Tree (TST) λ. The slicing algorithm (see Algorithm 2) is used to reduce λ to a reduced TST λ' w.r.t. an LTL$_{-X}$ formula ϕ. From the reduced TST λ', the TST2Text algorithm creates a textual expression which in turn will be deserialised and visualised by NOVA Workflow as a CWML model M'. The TST2Text algorithm is not included in the paper since it is straightforward. In Section 3.3, we prove that the models M and M' are stuttering equivalent w.r.t. ϕ.

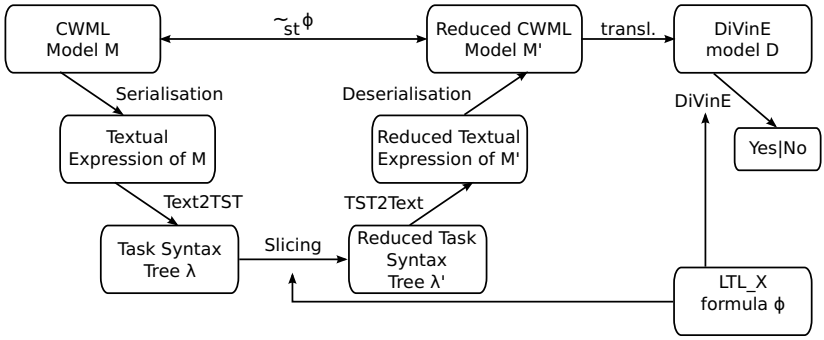


Fig. 1. Overview of the proposed approach

Section 2 provides some background information. In Section 3 we present the model slicing algorithm and the proof of the stuttering equivalence of a workflow model M and the corresponding reduced model M'. Section 4 presents a realistic experiment on a real world model for health services delivery. Section 5 relates our approach to other work, and Section 6 concludes the paper and outlines future research.

2 Preliminaries

A workflow model in CWML consists of tasks and operators connecting these tasks. The execution of some tasks are guarded by preconditions, and may perform some actions when executed. We begin by reviewing some definitions.

Definition 2.1 (Term, Precondition and Action) A term t is recursively defined using BNF as $t ::= c \mid \chi \mid t \circ_A t$, where $\circ_A \in \{+, -, *, \div\}$, c is a natural number and χ is a (natural) variable. A precondition is a formula ψ defined as $\psi ::= t \circ_C t \mid \psi \circ_P \psi$, where $\circ_C \in \{<, \leq, >, \geq, ==\}$ and $\circ_P \in \{\&\&, \|\}$. An action α is an assignment defined as $\alpha ::= v = t$; v is called an *assignee* variable. We abuse the notation $\{\alpha\}$ to denote a set of actions.

A compensable task can undo or rollback its operations at a later time after its successful execution, if required. In CWML, compensable tasks are composed

with t-calculus operators [11] to provide for a variety of methods of compensation. t-calculus allows one to combine compensable transactions to set up a long running business transaction which has compensation as its main error recovery technique. A compensable task consists of atomic compensable tasks composed with the compensable operators $\circ_{T_c}$.

Definition 2.2 (Compensable Task) A compensable task T_c is recursively defined using BNF as $T_c ::= \tau_c \mid (\{\psi_{T_{c1}}\}T_{c1} \circ_{T_c} \{\psi_{T_{c2}}\}T_{c2})$, where $\tau_c ::= \langle id\rangle\{\alpha_{\tau_c}, \alpha^b_{\tau_c}\}$ is an atomic compensable task which has a set of forward and compensation actions associated to it and $\langle id\rangle$ is the name of the task, moreover, $\circ_{T_c} \in \{\textcircled{\bullet}, \textcircled{\wedge}, \textcircled{\times}, \textcircled{S}, \textcircled{A}\}$ is a t-calculus operator. Only for $T_{c1}\textcircled{\times}T_{c2}$, does each of T_{c1}, T_{c2} have a precondition $\psi_{T_{c1}}, \psi_{T_{c2}}$, respectively.

The operators used to combine compensable tasks are explained as follows:

- $T_1\textcircled{\bullet}T_2$ (Sequential): T_1 will be executed first, then T_2 will be executed.
- $T_1\textcircled{\wedge}T_2$ (Parallel): T_1 and T_2 will be executed in parallel. If either of them is aborted, the other one will also be aborted.
- $T_1\textcircled{\times}T_2$ (Internal choice): Exactly one of the tasks will be executed.
- $T_1\textcircled{S}T_2$ (Speculative choice): T_1 and T_2 will be executed in parallel, the task that reaches the goal first will be accepted, the other one will be aborted.
- $T_1\textcircled{A}T_2$ (Alternative forwarding): T_1 will be executed first to achieve the goal, if T_1 is aborted, T_2 will be executed to achieve the goal.

A task may contain both compensable tasks and uncompensable tasks, connected by operators.

Definition 2.3 (Task) A task T is recursively defined using BNF as $T ::= \tau \mid T_c \mid (\{\psi_{T_1}\}T_1 \circ_T \{\psi_{T_2}\}T_2) \mid \{\psi_T\}(T)^+$, where $\tau ::= \langle id\rangle\{\alpha\}$ is an atomic uncompensable task which has a set of actions $\{\alpha\}$ associated to it and $\langle id\rangle$ is the name of the task, moreover, $\circ_T \in \{\bullet, \wedge, \times, \vee\}$ is a binary operator, and T^+ is a unary operator applied to T. Only for $T_1 \times T_2$, $T_1 \vee T_2$ and T^+, does each of T_1, T_2, T have a precondition $\psi_{T_1}, \psi_{T_2}, \psi_T$, respectively.

The operators used to combine tasks are explained as follows:

- $T_1 \bullet T_2$ (Sequential): T_1 will be executed first, then T_2 will be executed.
- $T_1 \wedge T_2$ (Parallel): T_1 and T_2 will be executed in parallel.
- $T_1 \times T_2$ (Exclusive choice): Exactly one of the two tasks will be executed.
- $T_1 \vee T_2$ (Choice): T_1 or T_2 or both will be executed in parallel.
- T^+ (Loop): T is executed once or some (specified) finite number of times as long as the precondition is true.

A workflow model M in CWML is a task with one input and one output condition. The underlying semantics of M is given by a Petri net; see [16] for the Petri nets underlying each of the tasks. The *state* of M is determined by the marking of its underlying Petri net. A path in M is defined as follows.

Definition 2.4 (Path) Given a workflow model M, a path $\pi = (s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots)$ is a sequence of states. The transition between one state and another caused by executing a task in M. The length of a path π is the number of state changes in π.

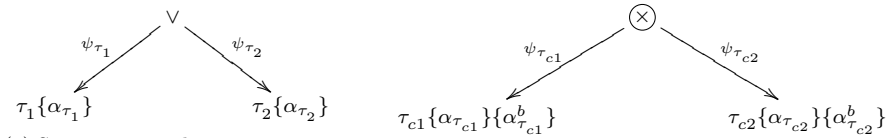
Note that by $(s_0 \rightarrow s_1 \dots s_j \xrightarrow{\tau_i} s_{j+1} \dots)$ we denote any path that involves the execution of the task τ_i. We use π_k to denote a path of length k. A task which causes a change of state could be either uncompensable or compensable. Therefore, in Section 3.3, we are able to prove the stuttering equivalence by using structural induction on the length of the path regardless of the kind of tasks used.

3 Workflow Slicing

In this section, we first present an algorithm to create TSTs from CWML models. Then we present our slicing algorithm and prove that using that algorithm to reduce a workflow model will yield a model that is stuttering equivalent to the original model.

3.1 Task Syntax Tree

Workflow models specified by CWML have a graphical structure. These structures are serialised as textual expressions which are valid w.r.t. the BNFs in Definitions 2.2 and 2.3. These expressions are parsed and represented as task syntax trees (TST) where a non-leaf node represents an operator and a leaf node represents an atomic (possibly compensable) task (see Fig. 2). Text2TST (see Algorithm 1), which outlines how a TST is generated from textual expressions, is adapted from the standard parsing process [12] with assignment of preconditions and actions. Note that only non-leaf nodes (operators) of $\times$ (XOR) or $\vee$ (OR) or $\otimes$ (Internal choice) or $^+$ have non-empty preconditions.



(a) Syntax tree with atomic uncompensable tasks

(b) Syntax tree with atomic compensable tasks

Fig. 2. Examples of workflow syntax trees

3.2 Slicing Algorithm

The slicing algorithm (see Algorithm 2) reduces the size of the workflow model based on the LTL formula subject to verification. Given a workflow, we reduce its TST λ based on the LTL formula ϕ that we wish to verify. In the workflow slicing algorithm, we first determine the variables occurring in ϕ and store them in a set E (Preserved Elements). Then, the set E will be extended recursively with tasks, preconditions, variables and actions that are *visible* w.r.t. the variables occurring

Input: Textual expression of workflow model M
Result: Task Syntax Tree λ

(i) Create tokens using Lexical Analysis;

(ii) Create parse tree, λ, using the grammar provided in Definitions 2.1, 2.2, and 2.3 (while parsing, parenthesis have precedence over operators);
  (a) Assign preconditions to the immediate branch(es) of non-leaf nodes;
  (b) Assign actions to leaf nodes (atomic tasks).

Algorithm 1. Text2TST: Constructs a TST from the textual expression of a workflow model

in ϕ. For example, a variable is visible and added to E, though it does not occur in ϕ, if it affects the preconditions for an operator which in turn affects the assignment of another variable in ϕ.

Definition 3.1 (Visible Precondition, Action, Atomic Task, and Operator) An action α is visible iff the *assignee* variable of α (see Definition. 2.1) is in E. An atomic (compensable) task τ is visible iff there exists any action α_τ of τ which is visible. An operator is visible iff any of its operands (can be operators or tasks) is visible. Preconditions of an operator are visible iff the operator is visible.

Suppose a variable v1 is in E, then an action v1 = 1 becomes a visible action since the variable v1 is the assignee of the action; i.e., it appears at the left hand side of the assignment operation. Note that if v1 appears on the right hand side of the assignment operation, it will not make the action visible, because in this situation the action will not affect the truth value of the property to be verified.

The slicing algorithm constructs a reduced syntax tree λ' by eliminating all nodes and preconditions not present in E. The next example illustrates the algorithm's main steps.

Example 3.2 (Workflow Slicing) Fig. 3 shows a workflow model M_{ex} containing 10 atomic tasks. The formula ϕ we wish to verify is: $G((\mathsf{v1} == 1) \rightarrow F(\mathsf{v2} == 1))$, meaning that if v1 is set with value 1, v2 will eventually be set with value 1. Task preconditions are shown along the edges and task actions are shown below the tasks. The textual representation of the lower portion of the workflow is as follows: (({} Task_2 { v1 = { 1,2 } } •({ v1 == 1 } Task_5 { v2 = 1 } ×{ v1 != 1 } Task_6 { v3 = 1 })) •{ } Task_9 { v6 = 1 }).

Fig. 4 shows the TST for M_{ex}. The variables v1 and v2 are visible since they appear in ϕ. The tasks Task_2, Task_5 and Task_3 become visible since the visible variables v1 and v2 appear as assignees in their actions. Paths from these tasks to the root node are then made visible and all the preconditions (v4 != 1, v4 == 1, v1 == 1 and v1 != 1) along these paths will become visible (indicated by solid lines). Since the preconditions v4 == 1 and v4 != 1 became visible, the variable v4 becomes also visible. Task_1 becomes visible since the visible variable v4 appears as assignee in its action. The rest of λ_{ex} will be invisible (indicated by dotted lines). Although the visible variable v1 appears on the right hand side of the action of Task_10, since the assignee variable v6 is not visible the action will not become visible.

Input: TST λ, and LTL formula ϕ
Result: Reduced TST λ'

$E \leftarrow \emptyset$;
for *each variable* $v \in \phi$ **do**
  /* add variables from the LTL formula */
  $E \leftarrow E \cup \{v\}$;
$size \leftarrow 0$;
while $size \neq E.size$ **do**
  $size \leftarrow E.size$;
  for *each leaf node* $\eta \in \lambda$ **do**
    if α_η *is visible* **then**
      $E \leftarrow E \cup \{\eta, \alpha_\eta\} \cup \alpha_\eta.variables$; $\eta_{curr} \leftarrow \eta.parentNode$;
        /* recursively add all ancestors */
      while η_{curr} *is not the root* **do**
        $E \leftarrow E \cup \{\eta_{curr}\}$;
          /* the four operators with conditions */
        if η_{curr} *is* $\times$ *or* $\vee$ *or* $\otimes$ *or* $^+$ **then**
          $\eta_l \leftarrow \eta_{curr}.leftChild$; $\eta_r \leftarrow \eta_{curr}.rightChild$;
            /* add all branch conditions */
          $E \leftarrow E \cup \{\psi_{\eta_l}, \psi_{\eta_r}\} \cup \psi_{\eta_l}.variables \cup \psi_{\eta_r}.variables$;
        $\eta_{curr} \leftarrow \eta_{curr}.parentNode$;

/* Construct the reduced syntax tree λ' by eliminating invisible elements from λ */
$\lambda' \leftarrow \lambda$;
for *each node* $\eta \in \lambda'$ **do**
  if $\eta \notin E$ **then**
    $\eta \leftarrow NIL$;

Algorithm 2. The slicing algorithm

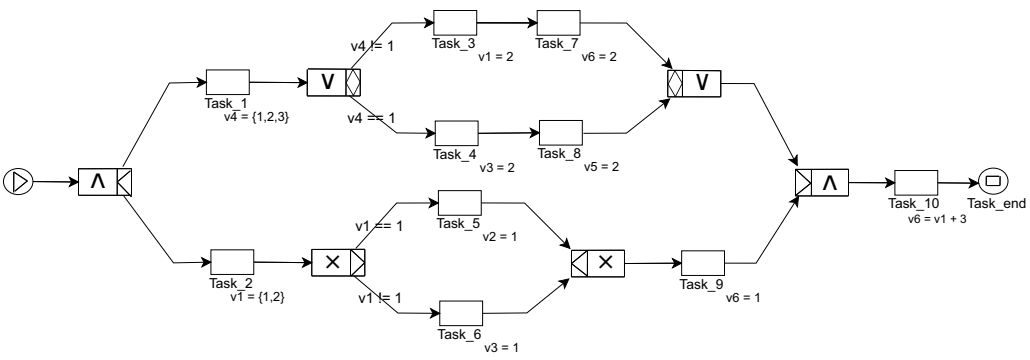


Fig. 3. A sample workflow M_{ex}

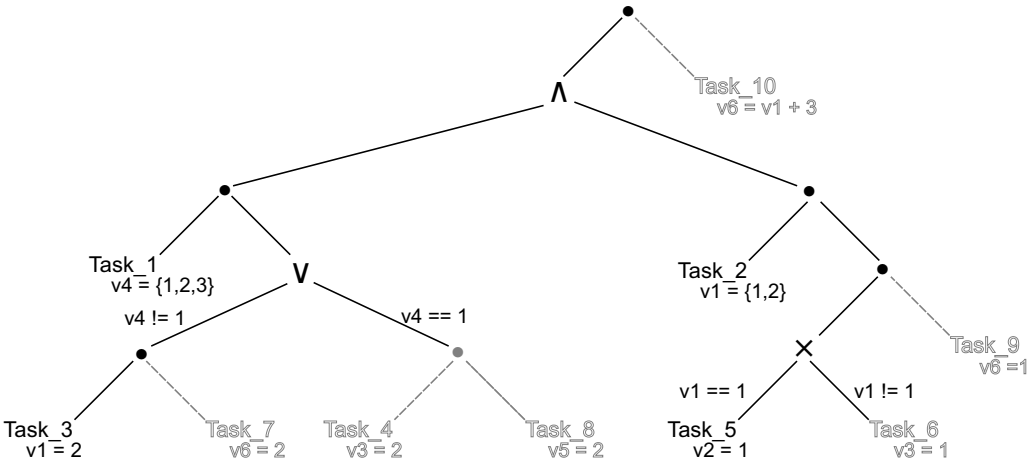


Fig. 4. The syntax tree λ_{ex} of the workflow M_{ex} from Fig. 3

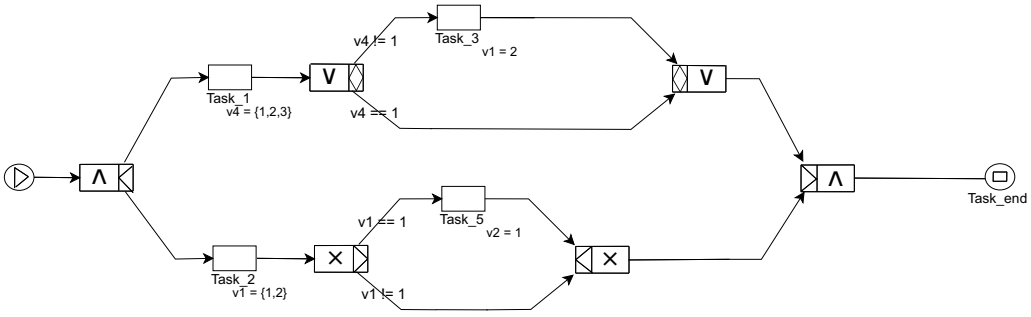


Fig. 5. The reduced workflow M'_{ex} of the workflow M_{ex} from Fig. 3

The reduced workflow M'_{ex} is shown in Fig. 5. M'_{ex} has fewer concurrent tasks but will provide the same verification result for ϕ (see the proof of stuttering equivalence in Section 3.3). In M'_{ex} the formula ϕ does not hold; one of the counter examples is the following sequence of task execution: (Task_2 → Task_1 → Task_3 → Task_9 → Task_end). This counter example shows that the variable v1 is set with the value 1 in task Task_2, and it is reset with another value 2 in task Task_3. For this execution the formula $G((\mathsf{v1} == 1) \rightarrow F(\mathsf{v2} == 1))$ does not hold in M'_{ex} and hence not in M_{ex}.

The complexity of the slicing algorithm is in the best case $O(n)$, where n is the number of atomic tasks; this happens when all tasks are invisible. In this situation, the algorithm checks every task once and then stops searching. The worst case complexity is $O(n^2 log\ n)$; this happens when the algorithm discovers a new visible element in every iteration.

3.3 Proof of Stuttering Equivalence

This section gives a proof for the stuttering equivalence of a workflow model M and the sliced model M' w.r.t. an LTL formula ϕ. We adapted the definition of stuttering equivalence from [4] using paths generated from the workflow model in

which each state results from the execution of one task and each transition denotes such an execution. Let V be the set of system variables, where the variables range over a finite set D, sometimes called the domain or universe of the interpretation. A *valuation* for V is a function that associates a value $d \in D$ to each variable $v \in V$. AVP is a set of *atomic valuation propositions*, where each proposition typically has the form $v = d$. Note that AVP is a subset of the atomic propositions AP. The labelling function $L : S \to 2^{AVP}$ returns the subset $L(s) \subseteq AVP$ which are true in $s \in S$. Further, the visible labelling function L_ϕ returns for each s the subset $L_\phi(s) \subseteq L(s)$ whose variables occur in ϕ.

Definition 3.3 (Visible Label Function) Let ϕ be an LTL formula and let V_ϕ be the set of variables occurring in ϕ; the visible label function for a state s, $L_\phi(s)$, is defined as $L_\phi(s) = \{p \mid var(p) \in V_\phi\}$, where p is a proposition in $L(s)$ and $var(p)$ returns the variable of proposition p.

We now define the stuttering equivalence of paths and workflow models.

Definition 3.4 (Stuttering Equivalence of Paths) Two finite paths $\pi = (s_0 \to s_2 \to s_3 \dots)$ and $\pi' = (s'_0 \to s'_2 \to s'_3 \dots)$ are stuttering equivalent w.r.t. an LTL formula ϕ, written $\pi \sim_{st^\phi} \pi'$, if there are two finite sequences of positive integers $0 = i_0 < i_1 < i_2 < \dots$ and $0 = j_0 < j_1 < j_2 < \dots$ such that for every $k \geq 0$, $L_\phi(s_{i_k}) = L_\phi(s_{i_k+1}) = \dots = L_\phi(s_{i_{k+1}-1}) = L_\phi(s'_{j_k}) = L_\phi(s'_{j_k+1}) = \dots = L_\phi(s'_{j_{k+1}-1})$.

Thus $\pi \sim_{st^\phi} \pi'$ iff the paths can be partitioned into finitely many blocks, such that the states in the kth block of π are labelled (w.r.t. ϕ) the same as the states in the kth block of π'.

Definition 3.5 (Stuttering Equivalence of Workflow Models) Two workflow models M and M' are stuttering equivalent ($M \sim_{st^\phi} M'$) w.r.t. an LTL formula ϕ iff:

- $L_\phi(s_0) = L_\phi(s'_0)$, where s_0, s'_0 are the initial states of M and M', respectively, i.e., M and M' have the same set of initial states (one each);
- for each path π of M there exists a path π' of M' such that $\pi \sim_{st^\phi} \pi'$; and,
- for each path π' of M' there exists a path π of M such that $\pi' \sim_{st^\phi} \pi$.

The following theorem shows that to prove that ϕ is invariant under stuttering, it is sufficient to show that M and M' are stuttering equivalent. The proof of the following theorem may be found in [4]. Note that $M, s_0 \models A\phi$ denotes that all paths in M starting at s_0 satisfy ϕ.

Theorem 3.6 *Any LTL$_{-X}$ formula is invariant under stuttering; that is, if ϕ is an LTL$_{-X}$ formula and $M \sim_{st^\phi} M'$ then $M, s_0 \models A\phi$ iff $M', s'_0 \models A\phi$.*

Lemma 3.7 *Given a workflow model M, an LTL$_{-X}$ formula ϕ and a reduced model M' generated by the slicing algorithm, for any two paths $\pi = s_i \xrightarrow{\tau_i} s_{i+1}$ and $\pi' = s'_j$, if τ_i is invisible and $L_\phi(s_i) = L_\phi(s'_j)$, then $L_\phi(s_i) = L_\phi(s_{i+1}) = L_\phi(s'_j)$ and hence $\pi \sim_{st^\phi} \pi'$.*

Proof. The proof follows from the fact that the invisible task τ_i will not change the truth value of any propositions in $L_\phi(s_i)$. □

Lemma 3.8 *Given a workflow model M, an LTL$_{-X}$ formula ϕ and a reduced model M' generated by the slicing algorithm, for any two paths $\pi = s_i \xrightarrow{\tau_i} s_{i+1}$ and $\pi' = s'_j \xrightarrow{\tau_i} s'_{j+1}$, if τ_i is visible and $L_\phi(s_i) = L_\phi(s'_j)$, then $L_\phi(s_{i+1}) = L_\phi(s'_{j+1})$ and hence $\pi \sim_{st^\phi} \pi'$.*

Proof. The proof follows from the fact that the visible task τ_i will have the same effect on both $L_\phi(s_i)$ and $L_\phi(s'_j)$. □

Theorem 3.9 *Given a workflow model M, an LTL$_{-X}$ formula ϕ and the reduced model M' generated by the slicing algorithm from M, then $M \sim_{st^\phi} M'$.*

Proof. Let ϕ be an LTL$_{-X}$ formula. The workflow model M is reduced to M' according to the slicing algorithm w.r.t. the formula ϕ. We will prove the theorem using structural induction on the length of paths in M and M'.

Base Case: Let π_0 be a path with 0 length in M; i.e., $\pi_0 = s_0$. We have to show that there exists a path π' in M' such that $\pi_0 \sim_{st^\phi} \pi'$.

Let $\pi'_0 = s'_0$ be a path in M' where s'_0 is the initial state of M'; according to the slicing algorithm, $L_\phi(s_0) = L_\phi(s'_0)$ as all the variables occurring in formula ϕ are visible and as a result they are preserved in M'. So, $\pi_0 \sim_{st^\phi} \pi'_0$.

Induction: Assume that for any path π_k in M, there exists a path π'_l in M', for an $l \leq k$, such that $\pi_k \sim_{st^\phi} \pi'_l$; that is, $s_0 \to s_1 \dots s_{k-1} \to s_k \sim_{st^\phi} s'_0 \to s'_1 \dots s'_{l-1} \to s'_l$. We have to show that, for any path $\pi_{k+1} = s_0 \to s_1 \dots s_k \xrightarrow{\tau_i} s_{k+1}$ in M, there exists in M' a path stuttering equivalent to π_{k+1}. There are two possibilities:

1. τ_i **is invisible** According to the slicing algorithm, τ_i is not present in M'. Due to the induction hypothesis, we have $s_0 \to s_1 \dots s_{k-1} \to s_k \sim_{st^\phi} s'_0 \to s'_1 \dots s'_{l-1} \to s'_l$; due to Lemma 3.7, $s_0 \to s_1 \dots s_{k-1} \to s_k \xrightarrow{\tau_i} s_{k+1} \sim_{st^\phi} s'_0 \to s'_1 \dots s'_{l-1} \to s'_l$. That is, the path π'_l in M' is stuttering equivalent to π_{k+1}.
2. τ_i **is visible** According to the slicing algorithm, τ_i is still present in M'. Due to the induction hypothesis, we have $s_0 \to s_1 \dots s_{k-1} \to s_k \sim_{st^\phi} s'_0 \to s'_1 \dots s'_{l-1} \to s'_l$; and due to Lemma 3.8, $s_0 \to s_1 \dots s_{k-1} \to s_k \xrightarrow{\tau_i} s_{k+1} \sim_{st^\phi} s'_0 \to s'_1 \dots s'_{l-1} \to s'_l \xrightarrow{\tau_i} s'_{l+1}$. That is, we have a path π'_{l+1} in M' that is stuttering equivalent to π_{k+1}.

□

Since the loops in our model are bounded loops, they eventually terminate and thus produce a finite number of states. We conclude that for any path in M there is a stuttering equivalent path in M'. Similarly we can prove that for any path in M' there is a stuttering equivalent path in M.

4 Experimental Results

The Canadian Hospice Palliative Care Association National Model (CHPCA 2002) [6] was built on an understanding of health, the illness and bereavement experiences, and the role hospice palliative care plays in relieving suffering and improving quality of life. We developed a Hospice Palliative Care (HPC) workflow, in collaboration with the local health authority the Guysborough Antigonish Strait Health Authority (GASHA), following the CHPCA 2002 model. This model contains general guidelines, called *Norms of Care*. We used the NOVA Workflow to model a HPC workflow and developed LTL formulae that the workflow must satisfy to comply with the norms.

After the patient's referral is received, her eligibility is checked for HPC. If eligible, the patient is sent for a set of therapeutic encounters which contains six essential steps – each is represented as a composite task in the "Overall" workflow (Fig. 6) – that guide the interaction between care givers, the patient and family. Fig. 6 also zooms into two composite tasks, namely PC_CONSULT and CARE_PLANNING. The palliative workflow has approximately 250 atomic tasks and 40 decision points. The PC_CONSULT task contains uncompensable tasks and the CARE_PLANNING task contains compensable tasks. Table 1 shows some preconditions and actions of tasks.

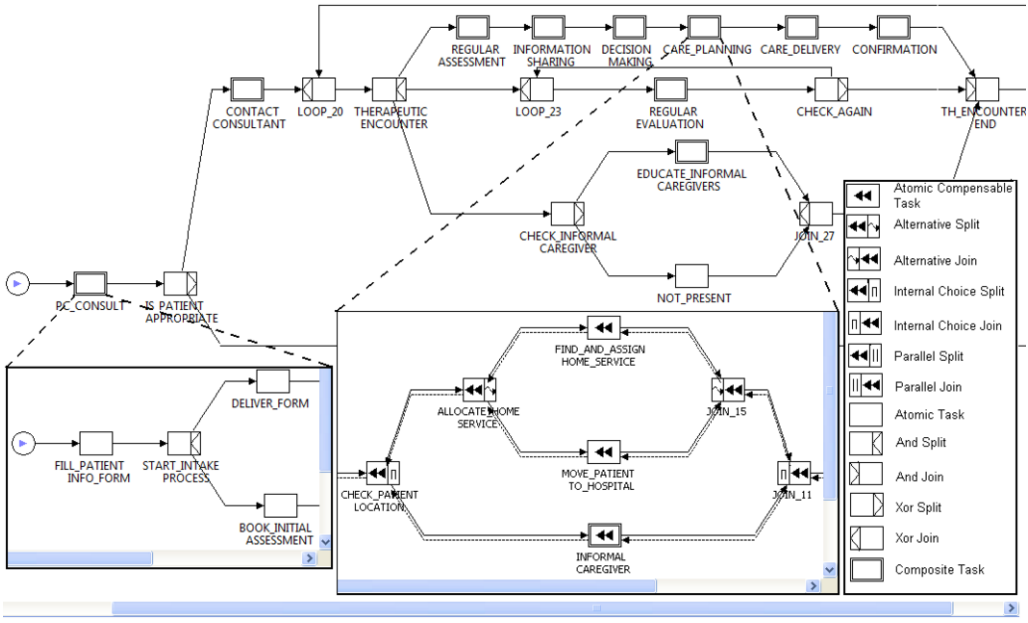


Fig. 6. Overview of the palliative care process model

Prop1 If patient is at home, then home service must be provided. Otherwise, the patient must move to the hospital (compensation via alternative choice). In LTL: G ((location == 1) → F ((home_service == 1) || (location == 2))). In Fig. 6, patient's location is set with either 1 (representing home) or 2 (representing hospital) at the time of registration (FILL_PATIENT_INFO_FORM); This information is accessed inside CARE_PLANNING workflow.

Table 1
Some of the tasks and their preconditions and actions from Fig. 6

Task	Preconditions, ψ	Actions, α
FILL_PATIENT_INFO_FORM		location = {1, 2}
ALLOCATE_HOME_SERVICE	location == 1	
INFORMAL_CAREGIVER	location == 2	informal_caregiver = 1
FIND_AND_ASSIGN_HOME_SERVICE		home_service = 1
ASSIGN_SOCIAL_WORKER	distressed == 1	social_worker = 1

Table 2
Verification results for the DiVinE model checker

Property	Holds?	WS + POR			POR		
		States	Mem (MB)	Time (s)	States	Mem (MB)	Time (s)
Prop1	Yes	126188210	88619.1	384.3	236576621	143836.2	1860
Prop2	Yes	107167421	83315.3	305.3	Unknown	Overflow	> hour
Prop3	Yes	128013744	88920.0	397.9	251323543	153290.3	1931
Prop4	Yes	127934841	88894.5	396.1	213254702	140215.0	1854
Prop5	Yes	13479	230.1	9.7	202233451	125804.1	1803

Prop2 If patient is distressed, then a social worker must be assigned in the care team. LTL: G ((distressed == 1) → F (social_worker == 1))

Prop3 If the patient is assigned a PPS of 50% or lower, s/he must be moved to the hospital. LTL: G ((pps ≤ 50) → F ((location == 2)))

Prop4 If the patient is with priority level of 3 or lower, s/he must be moved to the hospital. LTL: G ((patients_level ≤ 3) → F ((location == 2)))

Prop5 If the patient's mobility change is identified, a Physiotherapist is notified. LTL: G ((change_in_mobility == 1) → F (ack_physiotherapist == 1))

All experiments were executed with 64 CPU's and 3GB memory (per CPU) on the `Mahone2` cluster of ACEnet (www.ace-net.ca). The results are shown in Table 2. Time and memory are reduced using WS (workflow slicing) + POR (partial order reduction) compared to using POR alone. More details of these (and larger) experimental results may be found in [16].

We also perfomed a general performance comparison for different types of workflows. These experiments were done using DiVinE 2.4 on a single CPU with 3GB of Memory. Workflows consisting only of ∧, workflows with only × and workflows with only ∨ were considered; for each operator a different number of tasks were tested. For operator ∧, the LTL property we verified was whether two tasks, e.g. Task_1 and Task_2, can occur concurrently (i.e., G(Task_1_active && !Task_2_active)). Table 3 shows the number of transitions for the verification with WS and POR for various numbers of concurrent tasks. Note that we have proved this property by contradiction, so "Accepting Cycle" "YES" means Task_1 and Task_2 execute concurrently. The slicing algorithm excludes all tasks except visible tasks. So in the result, the number of states in WS + POR remains the same. Workflows with × were tested using various number of tasks, and it was determined whether Task_1 and Task_2 were mutually exclusive (In LTL_{-X}, G(Task_1_active → F(!Task_2_active) || G(Task_2_active → F(!Task_1_active)))). Workflows with operator ∨ were tested using various number of tasks, and it was determined whether the join task is eventually reachable (In LTL_{-X}, G F(Join_operator_active)). From these experimental

Table 3
Comparison for ∧ (AND), × (XOR) and ∨ (OR)

Case	Tasks	Acc Cycle	POR		WS + POR	
			States	Time (s)	States	Time (s)
∧ (AND)	5	YES	107	< 1	15	< 1
	10	YES	4396	< 1	15	< 1
	15	YES	48784	3	15	< 1
× (XOR)	5	NO	27	< 1	24	< 1
	10	NO	37	< 1	29	< 1
	15	NO	47	< 1	34	< 1
∨ (OR)	5	NO	99	< 1	1025	< 1
	10	NO	20197	3	1025	< 1

results we can see the effectiveness of the slicing; it becomes especially significant in situations where there are ∧ or ∨ with many tasks.

5 Related Work

Program slicing [21] is a well-studied technique. The basic idea is to abstract away variables and statements that do not influence the "point of interest", called the *slicing criterion*. Program slicing can be applied to debugging, testing, software maintenance, and formal verification. Hatcliff et al. [7] extract slicing criteria using primitive propositions in LTL formulae and more importantly, define and prove formally the *correctness* of program slicing.

Sloane and Holdsworth [20] propose *generalized slicing*, which deals with different kinds of software entities and constructs. More importantly (w.r.t. the relevance of our work), they use the program syntax tree as the vehicle for the slicing algorithm. Unfortunately they have not included formal verification in their framework. Millett and Teitelbaum [14] propose a slicing algorithm for Promela (the input language of the model checker SPIN). Barbuti et al. [2] present, from the model checking point of view, a general theoretical result of an equivalence between a transition system model and the reduced one based on formulae represented in their proposed temporal logic called the *selective mu-calculus*. All these works strongly suggest that slicing can be applied to modelling languages at different abstraction levels.

Slicing techniques have been applied to Petri nets. Evangelista et al. [5] present a reduction technique for Coloured Petri nets (CPN). This technique only preserves the liveness of the net and only those LTL formulae that do not observe the reduced transitions of the net. Rakow [18] presents a Petri net slicing algorithm and applies it to the verification of LTL formulae; the case study in the paper is based on a small textbook workflow example. We remark that CWML can be deemed as an abstraction of CPN. We required two distinct CPN one for the atomic uncompensable tasks and another for compensable tasks [17] as the two basic building blocks of the language and built up more complex Petri nets for each composite task. This type of abstraction is needed as real world workflows are generally complex and Petri net (including CPN) models of them can easily grow to be too large to be manageable.

There are several works that reduce the size of a workflow model. Wynn et al. [23] present reduction rules for YAWL [22] workflows with *Cancellation* regions and *OR-joins* to reduce the size of the workflow, while preserving its essential

formulae w.r.t. a particular analysis problem. There, the authors only focused on the reachability analysis, whereas our slicing method works for any LTL$_{-X}$ formula. Awad et al. [1] present a reduction procedure for BPMN graphs, but formal studies on the model equivalence are lacking. An ADEPT2 [19] workflow can be verified using the SeaFlows compliance checker [10]. In [10] authors discuss a data abstraction technique. As combining slicing and data abstraction is common practice in this area, we expect their technique and our slicing algorithm should complement each other well.

6 Conclusion and Future Work

This paper presents a model slicing algorithm. We prove the stuttering equivalence of the original models and the reduced models generated by the algorithm. This technique has been integrated into NOVA Workflow framework. Our experiments show that the technique greatly reduces the amount of memory and time for verification and makes verification of real world models of compensable systems possible.

We expect that translation of CWML to other model checkers is straightforward. Once other automated translation methods are developed, other model checkers can be used to verify large workflow models. In the future, we will consider time [13] in the model slicing algorithm as many specifications in a safety critical system such as healthcare are time sensitive. In [7] the authors defined *control dependence* which might be used in our approach to identify infinite loops from a model by preprocessing. In future we will enhance our work with unstructured workflow models with an infinite number of states.

Acknowledgement

This research is supported by Natural Sciences and Engineering Research Council of Canada, by an Atlantic Computational Excellence Network (ACEnet) Post Doctoral Research Fellowship and by the Atlantic Canada Opportunities Agency. The computational facilities are provided by ACEnet.

References

[1] Awad, A., G. Decker and M. Weske, *Efficient Compliance Checking Using BPMN-Q and Temporal Logic*, in: *BPM 2008: 6th International Conference on Business Process Management*, Lecture Notes in Computer Science **5240** (2008), pp. 326–341.

[2] Barbuti, R., N. D. Francesco, A. Santone and G. Vaglini, *Selective Mu-Calculus and Formula-Based Equivalence of Transition Systems*, Journal of Computer and System Sciences **59** (1999), pp. 537 – 556.

[3] Barnat, J., L. Brim, M. Češka and P. Ročkai, *DiVinE: Parallel Distributed Model Checker (Tool paper)*, in: *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)* (2010), pp. 4–7.

[4] Clarke, E. M., O. Grumberg and D. A. Peled, "Model Checking," The MIT Press, 1999.

[5] Evangelista, S., S. Haddad and J.-F. Pradat-Peyre, *Syntactical Colored Petri Nets Reductions*, in: *ATVA 2005: 3rd International Symposium on Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science **3707** (2005), pp. 202–216.

[6] Ferris, F. D., H. M. Balfour, K. Bowen, J. Farley, M. Hardwick, C. Lamontagne, M. Lundy, A. Syme and P. J. West, *A Model to Guide Hospice Palliative Care* (2002).

[7] Hatcliff, J., M. B. Dwyer and H. Zheng, *Slicing Software for Model Construction*, Higher Order and Symbolic Computation **13** (2000), pp. 315–353.

[8] IBM, Bea, Microsoft, SAP and Siebel, "Business Process Execution Language for Web Services Version 1.1," (2003).

[9] Ip, C. N. and D. L. Dill, *Better Verification Through Symmetry*, Formal Methods System Design **9** (1996), pp. 41–75.

[10] Knuplesch, D., L. T. Ly, S. Rinderle-Ma, H. Pfeifer and P. Dadam, *On enabling data-aware compliance checking of business process models*, in: *ER 2010: 29th International Conference on Conceptual Modeling*, Lecture Notes in Computer Science **6412** (2010), pp. 332–346.

[11] Li, J., H. Zhu, G. Pu and J. He, *A Formal Model for Compensable Transactions*, in: *ICECCS 2007: 12th IEEE International Conference on Engineering Complex Computer Systems* (2007), pp. 64–73.

[12] Louden, K. C., "Compiler Construction: Principles and Practice," Course Technology, 1997.

[13] Mashiyat, A. S., F. Rabbi and W. MacCaull, *Modeling and Verifying Timed Compensable Workflows and an Application to Health Care*, in: *Proceedings of FMICS 2011: 16th International Workshop on Formal Methods for Industrial Critical Systems*, Lecture Notes in Computer Science **6959** (2011), pp. 244–259.

[14] Millett, L. I. and T. Teitelbaum, *Slicing Promela and its Applications to Model Checking, Simulation, and Protocol Understanding*, in: *SPIN 1998: 4th International SPIN Workshop*, 1998, pp. 75–83.

[15] Peled, D., *Ten years of partial order reduction*, in: *CAV 1998: the 10th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science **1427** (1998), pp. 17–28.

[16] Rabbi, F., "Design, Development and Verification of a Compensable Workflow Modeling Language," Master's thesis, Dept. of Math., Stats. and CS, StFX University, Canada (2011).

[17] Rabbi, F., H. Wang and W. MacCaull, *Compensable Workflow Nets*, in: *Proceedings of ICFEM 2010: 12th International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science **6447** (2010), pp. 122–137.

[18] Rakow, A., *Slicing Petri Nets with an Application to Workflow Verification*, in: *SOFSEM 2008: 34th Conference on Current Trends in Theory and Practice of Computer Science*, Lecture Notes in Computer Science **4910** (2008), pp. 436–447.

[19] Reichert, M., S. Rinderle, U. Kreher, H. Acker, M. Lauer and P. Dadam, *ADEPT2 - Next Generation Process Management Technology*, in: *4th Heidelberg Innovation Forum* (2007).

[20] Sloane, A. M. and J. Holdsworth, *Beyond Traditional Program Slicing*, in: *ISSTA 1996: International Symposium on Software Testing and Analysis* (1996), pp. 180–186.

[21] Tip, F., *A Survey of Program Slicing Techniques*, Journal of Programming Languages **3** (1995), pp. 121–189.

[22] van der Aalst, W. M. P. and A. H. M. ter Hofstede, *YAWL: yet another workflow language*, Information Systems **30** (2005), pp. 245–275.

[23] Wynn, M. T., W. M. P. van der Aalst, A. H. M. T. Hofstede and D. Edmond, *Reduction Rules for YAWL Workflows with Cancellation Regions and OR-Joins*, Information and Software Technology **51** (2009), pp. 1010–1020.