

An Automated Translator for Model Checking Time Constrained Workflow Systems

Ahmed Shah Mashiyat, Fazle Rabbi, Hao Wang, and Wendy MacCaul

Centre for Logic and Information
St. Francis Xavier University
Antigonish, Canada
{x2008ooc,x2010mcf,hwang,wmaccaul}@stfx.ca

Abstract. Workflows have proven to be a useful conceptualization for the automation of business processes. While formal verification methods (e.g., model checking) can help ensure the reliability of workflow systems, the industrial uptake of such methods has been slow largely due to the effort involved in modeling and the memory required to verify complex systems. Incorporation of time constraints in such systems exacerbates the latter problem. We present an automated translator, *YAWL2DVE-t*, which takes as input a time constrained workflow model built with the graphical modeling tool YAWL, and outputs the model in DVE, the system specification language for the distributed LTL model checker DIVINE. The automated translator, together with the graphical editor and the distributed model checker, provides a method for rapid design, verification and refactoring of time constrained workflow systems. We present a realistic case study developed through collaboration with the local health authority.

Keywords: Workflow Systems, Modeling, Time, Automated Translation, Distributed Model Checking.

1 Introduction

Workflow Management Systems (WfMSs) improve business processes by identifying needs, reducing waste and duplication of work, ensuring completion of projects on time and in accordance with plans, improving efficiency, facilitating documentation, and creating solutions based on the analyzed process requirements. WfMSs such as Staffware, WebSphere MQ Workflow, FLOWer, SAP Workflow, YAWL are adopted widely in the industry because they facilitate the visualization and analysis of a business process. Many of today's workflows are complex requiring a high degree of flexibility, massive data and knowledge management, and complex timing [1]. However, the resulting implementations of unverified large and complex workflow models are at risk of undesirable runtime executions. Current WfMSs facilitate the enactment of workflows with some degree of fault-tolerance, e.g., exception handling, but formal verification capacity is limited.

Model checking is an automatic analysis method, which explores all possible states of a modeled system to verify whether the system satisfies a formally specified property. It is widely used in industrial applications, e.g., computer hardware and software, and has great potential for verifying models of complex and distributed business processes. Un-timed model checkers like SPIN and SMV can generally only represent and verify the *qualitative* temporal relations between events, which constrains their use for verifying real-time systems. *Timed* model checking, the method to formally verify real-time systems, is attracting increasing attention from both the model checking community and the real-time community. An extensive survey of formal methods for the specification and verification of timed systems in [2] contains references of over 200 publications. Despite the intensity of research dedicated to the specification and validation of real-time requirements, relatively little work has been done on formally modeling time-constrained workflows and their verification. When time becomes a factor in the activities running concurrently, the notion of time is required to precisely model in the workflow. *Quantified* time notions, including time instance and duration must be taken into account for timed model checking. For example in a safety critical application such as in an emergency department, after an emergency case arrives at the hospital, standard model checking can only verify whether “*The patient receives a certain treatment*”, but to save the patient’s life, it should be verified whether “*The patient receives a certain treatment within half an hour*”.

There are different approaches for modeling and verifying time constraints for workflow systems. Marjanovic et al. [3] provides a conceptual model to specify and verify timing aspects of production workflow; however, a production workflow lacks the notion of delay time between two consecutive tasks. Moreover, the verification is with respect to a chosen execution sequence. Many formalisms with time extensions have been presented as the basis for timed model checkers. Two popular ones, due to their simple graphical representations and solid mathematical formalisms, are: (1) *timed automata* [4], which is an extension of finite-state automata with a set of clock variables to keep track of time (which is under certain circumstances decidable); (2) *time Petri Nets* [5], which is an extension of Petri Nets with timing constraints on the firing of transitions. The later should be distinguished from *timed Petri Nets* which can store information on both arcs and tokens and are undecidable. A validation method for workflow specifications using UPPAAL (a timed automata based model checker) is presented in [6]. Models in a timed automata based model checker can not represent at which time instant a transition is executed within a time region; such model checkers can only deal with a specification involving a time region or a pre-specified time instant and cannot store the exact time instant at which a transition is executed. However, the *stop-watch* automata [7], an extension of timed automata, is proposed to tackle this; unfortunately, as Krcál and Yi discussed in [8], since the reachability problem for this class of automata is undecidable, there is no guarantee for termination in the general case. An approach for time constrained workflow verification using time Petri Nets can also be found in [9].

Time Petri Net tools such as Romeo [10] (which can verify a subset of Timed CTL), TINA [11] (which can verify LTL) can also be used for time constrained workflow verification.

The *state explosion* problem often limits the applicability of the above tools for real world workflow models. Distributed model checkers exploit the power of distributed computing facilities so that much larger memory is available to accommodate the state space of the system model; parallel processing of the states can, moreover, reduce the verification time [12]. For these reasons, we are particularly interested in the distributed LTL model checker DIVINE [13].

Previously, we manually translated a number of workflow patterns [14] into DVE, the modeling language of DIVINE. These patterns, defined in [15], are well accepted as the basic building blocks for the design and development of workflow models. Building a model, using these patterns, is tedious and error-prone; so this approach is not feasible for large and complex models. The tool *YAWL2DVE* which can automatically convert a YAWL model into DVE was presented in [16] but it was unable to handle time constraints.

Here, we present *YAWL2DVE-t*, which can automatically translate a graphical time constrained YAWL workflow model into DVE, thus reducing the difficulty of representing a time constrained workflow system in the input language of a model checking tool. Use of DIVINE enables us to handle the huge memory requirement for real world complex models. This approach enhances the “*push button technology*” of verification to a further step, allowing the users to model the system in a graphical language, such as YAWL, input a temporal property (with or without time constraints) and immediately do the model checking. Users with little expertise modeling with the model checking language can easily use it.

The remainder of this paper is as follows: Section 2 presents some background topics; Section 3 describes the modeling and verification method of time constrained workflow; Section 4 presents a case study and Section 5 concludes the paper and offers some directions for future work.

2 Preliminaries

This section provides background information about the tools used in this work. We begin by describing workflows and the workflow management system YAWL. Then we describe the DIVINE model checking tool and its modeling language.

2.1 Workflow and YAWL

For control purposes, workflow may be viewed as an abstraction of the real work under a chosen aspect that serves as virtual representation of the actual work. Therefore, a workflow is a collection of activities and the dependencies among those activities. The activities correspond to individual tasks in a business process. Dependencies determine the execution sequence of the activities and the data flow among these activities.

YAWL is a workflow management system, based on a concise and powerful modeling language. YAWL handles complex data, transformations, integration with organizational resources, and Web Service integration. YAWL uses a Petri net-based formalism extended with additional features to facilitate the modeling of complex workflows. A *workflow specification* in YAWL is a set of extended workflow nets (EWF-nets) which are made up of tasks, conditions and flow relations between them. A task (activity) is a description of a unit of work that may need to be performed as part of a workflow. The transfer of work between two tasks is done through a flow relation, which is depicted as unidirectional arrows in a YAWL model. In a YAWL model, every task must lie on a path from the start condition to the end condition. By default, a YAWL task can only have one outgoing flow and one incoming flow. When we need more outgoing flows from a task or incoming flows to a task, we have to use one of three kinds of split (for outgoing flows) and three corresponding kinds of join (for incoming flows); OR, XOR, and AND. The OR-Split is used to trigger some, but not necessarily all outgoing flows to other tasks. The XOR-Split is used to trigger only one outgoing flow. The AND-Split is used to start a number of task instances simultaneously. Corresponding XOR-Joins, OR-Joins, and AND-Joins are used to combine the incoming flows of a task. Tasks are either atomic or composite. Graphically in YAWL, an atomic task is represented by a rectangular box and a composite task is represented by a double rectangular box. Each task (either composite or atomic) can have multiple instances. Each atomic task can be assigned a timer called *Task Timeout*. It is also possible to set an activation type and an expiry value (Time units are Second, Minute, and Hour) for the timer. The timer can be activated either when a task is enabled or when it starts. A more detailed description of YAWL can be found in [17].

2.2 The DIVINE Model Checker and Its Modeling Language

DIVINE [18] is a distributed-memory explicit-state model checker, which employs the aggregate power of network-interconnected clusters to verify systems using distributed algorithms. DVE, the modeling language of DIVINE, is rich enough to describe systems made of synchronous and asynchronous processes communicating via shared memory and buffered or unbuffered channels. Like in Promela (the modeling language of SPIN), a model described in DVE consists of processes, message channels and variables. Each process, identified by a unique name, consists of a list of local variable declarations, process state declarations, initial state declaration and a list of transitions which start using the keyword *trans*. Variables can be global (declared at the beginning of DVE source code) or local (declared at the beginning of a process), they can be of `byte` or `int` type. A transition transfers a process from $stateid_1$ to $stateid_2$, the transition may contain a guard (which decides whether the transition can be executed), a synchronization (for communications between processes) and effects (which assign new values to local or global variables). Therefore, we have:

Transition ::= *stateid*₁ -> *stateid*₂ {Guard Sync Effect};

The **Guard** contains the keyword *guard* followed by a Boolean expression and the **Effect** contains the keyword *effect* followed by a list of assignments. The **Sync** follows the denotation for communication in the Communicating Sequential Processes (CSP) language, ‘!’ for sending and ‘?’ for receiving. The synchronization can be either asynchronous or rendezvous. Value(s) can be transferred in a channel identified by *chanid*. Declarations of channels follow declarations of global variables. Therefore, we have:

Sync ::= sync *chanid* ! SyncValue | *chanid* ? SyncVariable ;

Linear Temporal Logic (LTL) is a temporal logic which allows the specification of qualitative relationships between events. LTL has the following syntax given in Backus Naur form:

$$\phi ::= p | (\neg\phi) | (\phi \wedge \phi) | (\phi \vee \phi) | (X \phi) | (F \phi) | (G \phi) | (\psi U \phi)$$

where ϕ , ψ are formulas, and p is an atomic formula; $X \phi$ says that ϕ holds next time, $F \phi$ says that ϕ holds eventually, $G \phi$ says that ϕ holds globally, and $\psi U \phi$ says that ψ holds until ϕ holds.

In DIVINE, both the system model and the LTL formula are represented by automata. Then the model checking problem is reduced to detecting in the combined automaton graph whether or not there is an accepting cycle. If there is an accepting cycle, a counter example is produced. The model specification in DVE code is stored in a *.dve* file and the LTL properties are written in an *.ltl* file. DIVINE automatically generates a corresponding *property process* from the LTL formula, combines that process with the DVE code of the model, and produces a *.mdve* file. DIVINE uses identifiers to designate atomic formulas. For the simple clinical workflow in Fig. 1, let us assume we want to verify “In all cases a patient is released within 10 hours”. Recalling that LTL formulas are built from propositional formulas using X , F , G , and U , we write, G (start_reception -> F (finish_patientRelease \wedge timeRequiredToReleasePatient)). We define *timeRequiredToReleasePatient* as *timeDifference* <= 10, where 10 means ten hours, *timeDifference* is a variable in the DVE code which stores the time difference between the start time of *Reception* and the finish time of *Patient Release*. In section 3.3, we show how to store the start time and the finish time of a task.

3 Modeling and Verification of Time Constrained Workflow

In our approach, first a workflow is modeled with YAWL, and then the model is translated into the DVE model specification by *YAWL2DVE-t*. Combining an LTL property with the DVE model specification, the DIVINE model checker determines whether the LTL property holds or not. If the property does not hold, DIVINE gives a counter example.

3.1 Timing Constraints and Their Representation in YAWL

When we talk about activities (or tasks) and the dependencies among them, time plays an important role. Several explicit time constraints have been identified for time management of an activity [19]. *Duration* is the time span required to finish a task. *Forced start time* prohibits executing some tasks before that certain time. *Deadline* is a time based scheduling constraint which requires that a certain activity be completed by a certain time [20]. A constraint, which forces an activity to be executed only on a certain fixed date, is referred to as a *fixed date constraint*. *Delay* is the time duration between two subsequent activities. Besides these explicit time constraints, some time constraints follow implicitly from the control dependencies and activity durations of a workflow model. They arise from the fact that an activity can start only when its predecessor activities have finished. Such constraints are called the *structural time constraints* since they abide by the control structure of the workflow [19]. The concept of *relative constraint* which limits the time distance (duration) between the starting/ending instants of two non-consecutive workflow activities can also be found in [21]. Yet another kind of constraint is a *periodic constraint*, which represents a periodic time interval, during which an activity can be started, for example, a task can be executed between Monday and Saturday of every week [22].

The YAWL execution engine supports only the *duration* and *deadline* constraints. However, we are using YAWL’s graphical tool to model workflow for verifying properties. In modeling, we can assign a *delay* constraint in the control flow arc label between two consecutive tasks. Consequently we can model both *duration* and *delay* constraints which together are capable of modeling the timing aspects of almost any workflow [23]. *Duration* and *delay* are expressed in some basic time units and by integer value following the Gregorian calendar i.e., year, month, week, day, hour, and minute.

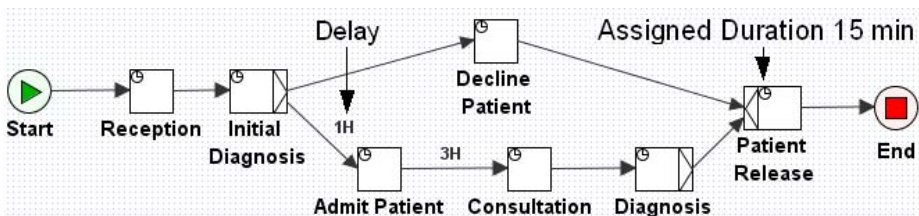


Fig. 1. Simple Clinical Workflow

In Fig. 1, *durations* and *delays* are shown in a simple clinical model designed by YAWL. Each task has a *duration* assigned to it. For example, *Patient Release* should be completed within fifteen minutes. Note that the *duration* assigned to each task is an integer value, thus constant. In real world, identifying a fixed *duration* for a particular task is unfeasible. Though we are assigning a constant value as *duration* of a task, the task can finish non-deterministically between

any time from zero to *duration*. So it suffices to identify the maximum possible *duration* for any task.

Delay time in the labels of the connector of two tasks means that the later task should wait until the *delay* time is elapsed after completing the earlier task. For example, after admission, patients have to wait three hours to get a *Consultation*. No *delay* time in the label indicates that a task initiates immediately after completing its preceding task(s). *Initial Diagnosis* should be performed without any delay after *Reception*. Neither *duration* nor *delay* is a mandatory attribute of a task.

3.2 Modeling Time in DVE

DIVINE is an un-timed model checker which generally can not verify timed systems. Lamport [24] advocated *explicit-time description methods* using a general model construct, e.g., global integer variables or synchronization between processes commonly found in standard un-timed model checkers, to realize timed model checking. He presented an explicit-time description method, which we refer to as LEDM, using a clock-ticking process (*Tick*) to simulate the passage of time, and a pair of global variables to store the lower and upper bounds of the time for each modeled system process. The method has been implemented with popular model checkers SPIN and SMV. Explicit-time description methods have three advantages: (1) they do *not* need specialized languages or tools for time description. Therefore, they can be applied in standard un-timed model checkers; (2) they enable the accessing and storing of the current time [25], a useful feature for the preemptive scheduling problems; and (3) they enable the use of large-scale distributed model checkers (e.g., DIVINE) for the timed model checking. Recently, Van den Berg et al. [26] successfully applied LEDM to verify the safety of the railway inter-locking for one of Australia's largest railway companies.

```

process Tick {
  state tick;
  init tick;
  trans
    tick -> tick { guard (all durationi && all delayi) > 0 &&
      (atleast one durationi || atleast one delayi) != INFINITY;
      effect now = now + 1,
      decrements all active durationi,
      decrements all active delayi; } ;
}

```

Fig. 2. *Tick* process in DVE

In LEDM [24], the current time is represented by a global variable *now* that is incremented by an added *Tick* process (See Fig. 2). As mentioned earlier, standard model checkers can deal with only integer variables, and a real-time

system can be modeled in discrete-time using an explicit-time description. So the *Tick* process increments *now* by 1. Note that in explicit-time description methods for standard model checkers, the real-valued time variables must be replaced by integer-valued ones. Therefore, these methods in general do not preserve continuous-time semantics; otherwise, an inherently infinite-state specification is produced and the verification is undecidable. However, these methods are sound for a commonly used class of real-time systems and their properties [27]. We believe workflow systems are ideally suited to this kind of analysis.

By assigning duration to a task and delay between two consecutive tasks as timing constraints, we can model time in a workflow system. A duration timing constraint forces the process, once initiated, to be finished within that time span. A delay constraint prohibits a task from being started, until the delay time has elapsed, counting the time from which the previous task has finished. Each system process P_i has two count-down timers, denoted as the global variable $duration_i$ and $delay_i$. A large enough integer constant, denoted as INFINITY, is initially assigned to the all $duration_i$ and $delay_i$ variables. Duration and delay timers with the value of INFINITY are not active and the *Tick* process will not decrement them. As *now* is incremented by 1, each non-INFINITY $duration_i$ and $delay_i$ is decremented by 1. Every model specification has a *Tick* process to simulate the time.

We observe that the value of *now* is limited by the size of type `integer` and careless incrementation can cause overflow error. This can be avoided by incrementing *now* using modular arithmetic, i.e., setting $now = (now + 1) \bmod \text{MAXIMAL}$ (MAXIMAL is the maximal integer value supported by the model checker). The limit of the maximum can be increased by linking several integers, i.e., when $(int_1+1) \bmod \text{MAXIMAL}$ becomes zero, int_2 increments by 1, and so on.

3.3 Workflow Tasks in DVE

In this paper, we map a task (either composite or atomic) in YAWL to a DVE process; multiple instances of a task are mapped to multiple instances of the same process. Control flow paths are mapped to DVE channels and messages between processes are represented, without loss of granularity, by integers, in DVE. A task is enabled after the completion of its preceding task(s). The split and join structures of YAWL, introduced in section 2.1, are identified in the translation process and handled using different algorithms. Details of these algorithms can be found in our previous work [16]. In Fig. 3, we present a DVE process for a YAWL task.

Initially, all processes are in the *idle* state which indicates that tasks are not ready to start (In Fig. 3 process P_i is in *idle* state). An incoming message, through channel *ChanPrevProcess*, from a previous task activates the process P_i . A transition from the *idle* state to the *waiting* state represents that the task is initialized and a *delay* is assigned for the task as an effect. If the *delay* time for the task is zero then it can commit another transition where the *duration* of a task is assigned as effect. If the *delay* time for the task is not zero then the task has to wait until the *delay* time becomes zero. The passing of time is simulated


```

process  $P_i$  {
  state ..., idle,waiting, working;
  init idle;
  trans
    idle -> waiting { sync ChanPrevProcess?;
                      effect set  $delay_i$ , set  $start_{P_i}$  to now;},
    waiting -> working { guard  $delay_i=0$ ; effect set  $duration_i$ ;},
    ... -> ... ,
    working -> idle { sync ChanNextProcess! ;
                     effect set  $duration_i$  to INFINITY,
                           set  $delay_i$  to INFINITY ,
                           set  $finish_{P_i}$  to now; };
}

```

Fig. 3. System process P_i in DVE

by the *tick* process (See Fig. 2). The task can be finished anytime within the *duration*. The transition from the *working* state to the *idle* state indicates that the task is finished. A message is sent through channel *ChanNextProcess* to activate the next process. As an *effect* of the transition the *duration* and *delay* time is set to INFINITY for that process. If any of the *duration* and *delay* timers is equal to zero, the transition in the *Tick* process is disabled. This forces the transition from the *working* state to the *idle* state for that process, if it is the only transition possible at this time. In this way, the *duration* and *delay* constraints are realized. To verify timed properties, we have to store the start time and finish time of all the processes. In this regard, for each process there are two global variables that store the start and end time. In Fig. 3, $start_{P_i}$ stores the start time and $finish_{P_i}$ stores the end time of process P_i .

3.4 YAWL2DVE-t: An Automated Translator

YAWL2DVE-t is an automated translator developed using Java, which can translate any workflow modeled using YAWL. It takes a YAWL (in XML) file as input and generates a *mdve* file as output; the *mdve* file can be combined with an *ttl* property file (may contain more than one property) and produce *dve* output file(s) (one for each property) which can be used for verification. The following steps are used in the translation by *YAWL2DVE-t*:

1. Parse XML and construct workflow components with timing information
2. Create links and channel numbers
3. Process multiple tasks
4. Generate DVE code from root net decomposition

For each task in the workflow model, *YAWL2DVE-t* will produce DVE code as described in section 3.3 . A part of the XML file for the simple clinical workflow in Fig. 1 is given in Fig. 4. The task ID's in the XML file are unique. An index and these ID's are used to generate unique channels for communication. In the

```

....
<task id="Consultation_7">
  <name>Consultation</name>
  <flowsInto>
    <nextElementRef id="Diagnosis_242" />
  </flowsInto>
  <join code="xor" />
  <split code="and" />
  <timer>
    <trigger>OnEnabled</trigger>
    <duration>PT5H</duration>
  </timer>
....

```

Fig. 4. Segment of XML File for Simple Clinical Workflow

DVE translation of the simple clinical workflow of Fig. 1, *YAWL2DVE-t* will create nine processes, one each for the *start* and *end* conditions, and seven for the tasks in between. The processes for the *start* and *end* conditions do not have any *delay* and *duration*, thus those processes are instantaneous. The *InputCondition_1* process (for *start* condition) will send a synchronization signal through the channel '*chan_InputCondition_1_0*'. The *Reception* process will be activated after receiving that signal and after completing its work (after the assigned *duration*) it will send a synchronization signal to the *Initial_Diagnosis* process through the channel '*chan_Reception_1*'. The *Initial_Diagnosis* process has an XOR-Split structure and can choose any of the channels non-deterministically to send a synchronization signal to the *Admit Patient* process or the *Decline Patient* process and the flow continues. The duration is stored in a *duration* tag in the XML file (*Consultation* task is assigned 5 hours as duration, see Fig. 4). The processes will be executed according to the workflow order, the next task reference is stored in a *nextElementRef* tag in the XML file. For each Net Decomposition (Composite task) of the workflow, a *NetDecomposition* instance will be created which contains one *InputCondition*, one *EndCondition*, and one or more task and condition instances.

A more detailed description (with algorithms) of step 1-4 can be found in our previous work [16].

4 Case Study and Property Verification

In this section we will examine how the above method can be useful for the verification of a workflow model with timing information. We study a Hospice Palliative Care workflow, which we are developing for the local health authority, the Guysborough Antigonish Strait Health Authority (GASHA), following the national Hospice Palliative Care model [28]. A number of sample specification properties are verified on the workflow model and a top level graphical view of the model, built with the YAWL modeling tool, is given in Fig. 5. Tabs across the

top refer to composite tasks each of which give rise to a subnet. Subnets may also contains composite tasks. See the appendix for a more detailed view of the model.

4.1 A Hospice Palliative Care Workflow

Palliative Care refers to the medical or comfort care that reduces the severity of a disease or slows its progress rather than providing a cure. For incurable diseases, in cases where the cure is not recommended due to other health concerns, and when the patient does not wish to pursue a cure, palliative care becomes the focus of treatment. For example, if surgery cannot be performed to remove a tumor, radiation treatment might be tried to reduce its rate of growth, and pain management could help the patient manage physical symptoms. The term “Hospice Palliative Care” was coined to recognize the convergence of hospice and palliative care into one movement that has the same principles and norms of practice [28].

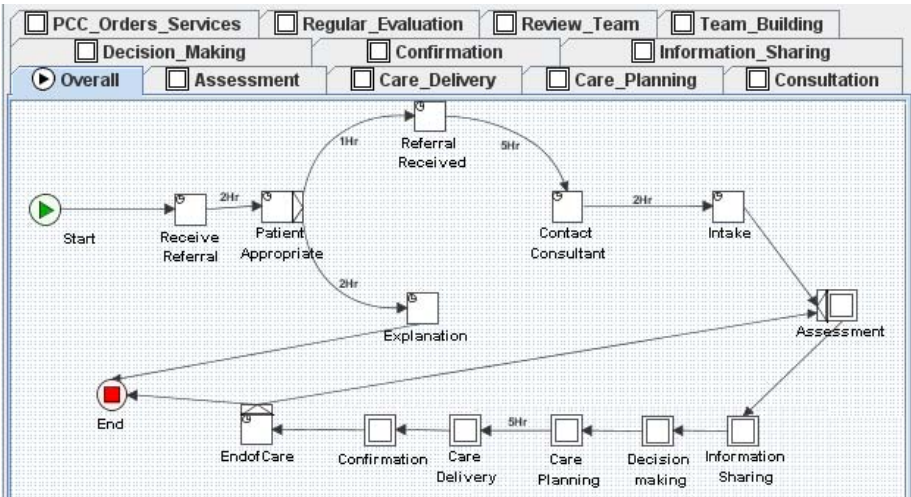


Fig. 5. Hospice Palliative Care Workflow Model for GASHA

The national model was developed to guide both the process of providing care to the patients and their families, and the development and function of hospice palliative care organizations. The model provides guidelines which they refer to as “norms of care” for quality of service, such as, “Requests for initial evaluation and ongoing follow-up are responded to within acceptable time frames”. Based on this norm, an organization will develop a more specific standard of practice that will establish the minimum requirements to be met at all times [28], such as:

- “Requests for initial evaluation are responded to within 48 hours.”
- “Requests for ongoing follow-up are responded to within 12 hours.”

The timing information in this model is for illustration purpose, so it does not reflect the actual time in the real world. We are working with GASHA to greatly refine the details of the process of care and include information on time, access control and other process specific information. A pilot study is underway to transform the documentation from a paper based format to an electronic version; details gleaned from this pilot will substantially enhance the model.

The Hospice Palliative Care process involves six essential and several basic steps that guide the interaction between care givers, and the patient and family. After the patient referral is received, it is determined whether the patient is eligible for Hospice Palliative Care. If the patient is not eligible, the workflow will end with proper explanation. Otherwise the patient is sent for the next set of care tasks. A consultant will collect the medical information from the patient, identify his priorities, and determine whether the patient requires a consultation with physician(s). This completes the registration process and a profile is created and the patient enters into a iterative process called a therapeutic encounter. The next six tasks are essential and must be completed during each encounter. Each of them is represented as a composite task and has tasks (atomic or composite) in its ‘Net-Decomposition’.

This model produces 144 processes with more than two thousands lines of code in DVE. Once the LTL properties are identified, they can be verified against the model (in DVE). If a property does not hold, the DIVINE model checker will produce a counter example. This approach greatly reduces the effort for modeling and rapid refactoring of a system model for verification. All we have to do is make a graphical representation of the workflow in YAWL, use the translator to get the DVE model for distributed verification and run DIVINE.

Property verification: Some properties of the Palliative Care model have been checked by the DVE model. The properties are Categorized into three groups: *safety properties*, *liveness properties*, and *time properties*. A *Safety property* states that the property must be true for all paths, informally, “Some bad thing never happens”. A *Liveness property* ensures the progress of the workflow, informally, “Some good thing will eventually happen”. *Time properties* refer to the properties that are related to time constraints. In the following, we articulate some norms described in [28] and their corresponding LTL formulas, the first two properties are *safety properties* and *liveness properties* respectively, the last four properties are *time properties*; 3 and 6 involve duration constraints, 4 and 5 involve both delay and duration constraints.

1. Limits of confidentiality are defined by the patient before information is shared.

$$G \neg(\text{confidentiality_not_defined} \wedge \text{information_shared})$$
2. End state is reached in all paths

$$G F \text{c_end}$$

3. Any errors in therapy delivery are reported to supervisors immediately. The standard acceptable time allowed for such reports is two hours.
 $G(\text{start_error_in_therapy} \rightarrow F(\text{finish_report_to_supervisor} \wedge \text{timeRequiredToFinish2Hours}))$
4. Requests for initial evaluation are responded to within 48 hours.
 $G(\text{start_initial_evaluation_request} \rightarrow F(\text{finish_initial_evaluation_respond} \wedge \text{timeRequiredToFinish48Hours}))$
5. Care planning should not take more than 1 day in each therapeutic encounter.
 $G(\text{start_care_planning} \rightarrow F(\text{finish_care_planning} \wedge \text{timeRequiredToFinish1Day}))$
6. Interview with the family member should be done within 1 hour.
 $G(\text{start_interview} \rightarrow F(\text{finish_interview} \wedge \text{timeRequiredToFinish1Hour}))$

In the verification process, we are simulating the time by a *Tick* process which increments the *now* variable. Therefore, to identify a relative time distance between two non-consecutive tasks we have to store the start time of the preceding task and the end time of the subsequent task in the DVE model; the method is described in section 3.3.

Some requirements of the Palliative Care model norms cannot be represented due to the limitations of the modeling language and LTL. For example, an LTL property cannot represent that “the information is as accurate as possible”. We are studying different methods to extend the language and the specification logic so that we can accurately specify and verify properties corresponding to real world requirements.

In our method, we are simulating time as states. *Delay* and *duration* can be expressed in time units, such as second, minute, hour, etc. While translating the YAWL model to the DVE code, we convert all the time units to the lowest one, which might blow up the state space to the workflow models where the time units of *delay* and *duration* vary substantially. To address this issue, we are incorporating the efficient EDM we proposed in [25], in which the *Tick* process may *leap* multiple time units in a tick.

Table 1. Experimental Results

Property	True/False	OWCTY/MAP	Time (s)	# of States	Memory (MB)
1	True	OWCTY	216.3	166914735	65062.6
2	True	OWCTY	1104.8	223439236	86953.3
3	False	MAP	6.7	957694	16262.5
4	True	OWCTY	303.9	217528767	80829.6
5	False	MAP	12.9	1557694	21246.3
6	True	OWCTY	312.4	218428214	81013.7

Table 1 gives the experimental results which shows that properties 1, 2, 4 and 6 holds in the model and properties 3 and 5 do not hold. The counter example for property 3 enables us to determine that the assigned duration for the

task *Report to supervisor* is not at an acceptable level. The counter example for property 5 helps us to find an execution sequence for which the *Care Planning* takes more than 1 day, refactor the model and redo the model checking. All experiments are executed on the Mahone2 cluster of ACEnet [29], the high performance computing consortium for universities in Atlantic Canada. The cluster is a Parallel Sun x4100 AMD Opteron (dualcore) cluster equipped with Myri-10G interconnection. Parallel jobs are assigned using the Open MPI library. We have used 32 CPUs and 5G memory per CPU for all these experiments. In the cluster, there are 139 nodes, each containing 16G of memory. Future models will be larger and will require more resources. Current model utilizes approximately 30% of the resources of the cluster.

Two model checking algorithms in DiVINE 0.8.3 are used, namely, *One Way to Catch Them Young* (OWCTY) and *Maximal Accepting Predecessors* (MAP). If the property of a model is expected to hold and the state space can fit completely into the (distributed) memory, OWCTY is preferable as it is three times faster than MAP to explore the whole state space. On the other hand, MAP can generally find a counterexample (if it exists) much more quickly as it works on-the-fly. For each property, we only show the better result of these two algorithms.

5 Conclusion and Future Work

This research is a part of an ambitious research and development project, Building Decision-support through Dynamic Workflow Systems for Health Care [30] among researchers at StFX in a collaboration with the local health authority GASHA and an industrial partner, Palomino System Innovations INC. In prior work, Miller and MacCaull [31] developed a prototype tableau-based model checker based on timed *BDI_{CTL}* logic, Hao and MacCaull [25,32] developed several new Explicit-time Description Methods. These efforts facilitate the verification of properties written in an extended specification language involving information such as real time and agents' beliefs, goals and intentions, in large workflow models.

In this paper, we described an automated translator that translates time constrained workflow models into DVE code. Using an automated translator will greatly reduce the cost and time for the verification and will make the verification of real world models possible. We are developing a translator to the input language of SMV and other model checkers for CTL model checking. Other enhancements for model checking, like data aware verification techniques, reduction technique for the state space of the model and verification of compensable transactions for workflow systems are also in progress. Real world workflow processes can be highly dynamic and complex in a health care setting. Verification that the system meets its specifications is essential for such a safety critical process and can save time, money, or even lives.

Acknowledgment. This research is sponsored by Natural Sciences and Engineering Research Council of Canada (NSERC), by an Atlantic Computational

Excellence Network (ACEnet) Post Doctoral Research Fellowship and by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund. The computational facilities are provided by ACEnet. We also like to thank Prof. Mary Heather Jewers from StFX School of Nursing, Jay Crawford, Keith Miller, Nazia Leyla, Zahidul Islam, Madonna MacDonald (VP, Community Service at GASHA) and numerous clinicians from GASHA.

References

1. Miller, K., MacCaull, W.: Toward web-based careflow management systems. *Journal of Emerging Technologies in Web Intelligence (JETWI) Special Issue: E-health Interoperability 1(2)*, 137–145 (2009)
2. Wang, F.: Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE 92*, 1283–1305 (2004)
3. Marjanovic, O.: Dynamic verification of temporal constraints in production workflows. In: *ADC '00: Proceedings of the Australasian Database Conference*, Washington, DC, USA, pp. 74–81. IEEE Computer Society, Los Alamitos (2000)
4. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* 126(2), 183–235 (1994)
5. Merlin, P.M.: A study of the recoverability of computing systems. PhD thesis, Department of Information and Computer Science, University of California, Irvine, CA (1974)
6. Gruhn, V., Laue, R.: Using timed model checking for verifying workflows. In: Cordeiro, J., Filipe, J. (eds.) *Computer Supported Activity Coordination*, pp. 75–88. INSTICC Press (2005)
7. Abdeddaïm, Y., Maler, O.: Preemptive job-shop scheduling using stopwatch automata. In: Katoen, J. P., Stevens, P. (eds.) *TACAS 2002*. LNCS, vol. 2280, pp. 113–126. Springer, Heidelberg (2002)
8. Krcál, P., Yi, W.: Decidable and undecidable problems in schedulability analysis using timed automata. In: Jensen, K., Podelski, A. (eds.) *TACAS 2004*. LNCS, vol. 2988, pp. 236–250. Springer, Heidelberg (2004)
9. Foyo, P.M.G.D., Silva, J.R.: Using time petri nets for modeling and verification of timed constrained workflow systems. In: *ABCM Symposium Series in Mechatronics*, vol. 3, pp. 471–478 (2008)
10. Gardey, G., Lime, D., Magnin, M., Roux, O.H.: Roméo: A tool for analyzing time petri nets. In: Etessami, K., Rajamani, S.K. (eds.) *CAV 2005*. LNCS, vol. 3576, pp. 418–423. Springer, Heidelberg (2005)
11. Berthomieu, B., Vernadat, F.: Time petri nets analysis with tina. In: *3rd International Conference on The Quantitative Evaluation of Systems (QEST 2006)*, pp. 123–124. IEEE Computer Society Press, Los Alamitos (2006)
12. Barnat, J., Brim, L., Ročkai, P.: Scalable multi-core ltl model-checking. In: Bošnački, D., Edelkamp, S. (eds.) *SPIN 2007*. LNCS, vol. 4595, pp. 187–203. Springer, Heidelberg (2007)
13. Barnat, J., Brim, L., Ročkai, P.: DiVinE 2.0: High-Performance model checking. In: *International Workshop on High Performance Computational Systems Biology (HiBi 09)*, pp. 31–32. IEEE Computer Society, Los Alamitos (2009)
14. Leyla, N., Mashiyat, A., Wang, H., MacCaull, W.: Workflow verification with divine. In: *8th International Workshop on Parallel and Distributed Methods in verification (PDMC '09)* (2009) Work in progress report

15. Russell, N., Hofstede, A.T., Aalst, W.M.P.V.D., Mulyar, N.: Workflow control-flow patterns: A revised view. Technical report, BPMcenter.org (2006)
16. Rabbi, F., Wang, H., MacCaull, W.: YAWL2DVE: An automated translator for workflow verification. In: 4th IEEE International Conference on Secure Software Integration and Reliability Improvement, pp. 53–59. IEEE Computer Society, Los Alamitos (2010)
17. Aalst, W.M.P.V.D., Hofstede, A.T.: Yawl: yet another workflow language. *Information Systems* 30(4), 245–275 (2005)
18. DiVinE Project, <http://divine.fi.muni.cz/> (last accessed June 2010)
19. Li, W., Fan, Y.: A time management method in workflow management system. In: Workshops at the Grid and Pervasive Computing Conference, pp. 3–10. IEEE Computer Society, Los Alamitos (2009)
20. WfMC: Workflow management coalition terminology and glossary. Technical Report (wfmc-tc-1011) v3.0. Technical report, Winchester, UK (1999)
21. Combi, C., Posenato, R.: Controllability in temporal conceptual workflow schemata. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 64–79. Springer, Heidelberg (2009)
22. Combi, C., Gozzi, M., Juárez, J.M., Oliboni, B., Pozzi, G.: Conceptual modeling of temporal clinical workflows. In: TIME, pp. 70–81. IEEE Computer Society, Los Alamitos (2007)
23. Li, H., Yang, Y.: Verification of temporal constraints for concurrent workflows. In: Yu, J.X., Lin, X., Lu, H., Zhang, Y. (eds.) APWeb 2004. LNCS, vol. 3007, pp. 804–813. Springer, Heidelberg (2004)
24. Lamport, L.: Real-time model checking is really simple. In: Borriore, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 162–175. Springer, Heidelberg (2005)
25. Wang, H., MacCaull, W.: An efficient explicit-time description method for timed model checking. In: 8th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 09), Eindhoven, The Netherlands, EPTCS, vol. 14, pp. 77–91 (2009)
26. Berg, L.V.D., Strooper, P.A., Winter, K.: Introducing time in an industrial application of model-checking. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 56–67. Springer, Heidelberg (2008)
27. Henzinger, T.A., Manna, Z., Pnueli, A.: What good are digital clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
28. Ferris, F.D., Balfour, H.M., Bowen, K., Farley, J., Hardwick, M., Lamontagne, C., Lundy, M., Syme, A., West, P.: A model to guide hospice palliative care: Based on national principles and norms of practice. Canadian Hospice Palliative Care Association, Ottawa (2002)
29. Atlantic Computational Excellence Network (ACENet), <http://www.ace-net.ca/> (last accessed, June 2010)
30. Centre for Logic and Information. St. Francis Xavier University, <http://logic.stfx.ca/> (last accessed, June 2010)
31. Miller, K., MacCaull, W.: Verification of careflow management systems with timed *BDI_{CTL}* logic. In: 3rd International Workshop on Process-oriented Information Systems in Healthcare (ProHealth'09), in Conjunction with BPM'09. LNBIP, vol. 43, pp. 623–634. Springer, Heidelberg (2010)
32. Wang, H., MacCaull, W.: Verifying real-time systems using explicit-time description methods. In: Workshop on Quantitative Formal Methods: Theory and Applications, Eindhoven, The Netherlands, EPTCS, vol. 13, pp. 67–79 (2009)