# Model Driven Workflow Development with $T_\square$

Fazle Rabbi and Wendy MacCaull

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

**Abstract.** Model Driven Engineering (MDE) refers to the systematic use of models as primary engineering artifacts throughout the engineering lifecycle. MDE has a lot of potential to make adaptive software systems, but it requires maturity and tool support. Here we present a domain specific language, called $T_\square$ (pronounced as T-Square) for writing workflow process specifications which allows us to write procedural statements for tasks and branch conditions, to query an ontology and to declare user interfaces. We apply transformation methods to generate executable software from the abstract process specifications.

**Keywords:** Workflow Management System, Model Driven Engineering, Ontology, Domain Specific Language, Adaptive System.

## 1 Introduction

Software researchers and developers require abstractions of their system to help them program in terms of their design intent rather than in terms of the underlying computing environments e.g., CPU, memory, and network devices [18]. Although early programming languages, such as assembly and Fortran, raised the level of abstraction by shielding developers from complexities of programming with machine code, they still had distinct "computing oriented" focuses. Advances in languages and platforms during the past two decades have minimized the need to reinvent common and middleware services, such as transactions, discovery, fault tolerance, event notification, security, and distributed resource management, by providing libraries, APIs, etc. But programmers and developers still need to focus on the use of those libraries, APIs, services, etc. Model Driven Engineering (MDE) further raises the level of abstraction in program specification and aims to increase automation in software development [18]. MDE offers a promising approach to alleviate the complexity of platforms by expressing domain concepts effectively by models. A model is specified by modeling notations or modeling languages. Since modeling languages are usually tailored to a certain domain, such a language is often called a Domain Specific Language (DSL). A DSL can be visual (e.g., UML, BPMN [1]) or textual (e.g., CSS, regular expressions, ant, SQL). DSL helps developers focus on a problem domain rather than on technical details [14].

Today business process models are frequently used to describe the behaviour of large systems with characteristics like concurrency, resource sharing, and synchronisation [11]. Many of today's workflows are complex requiring a high degree of flexibility, and massive data and knowledge management. In this paper we present a DSL called $T_{\Box}$ (pronounced T-Square) for writing workflow process specifications, which incorporates the following features: a) a simple syntax for i) writing procedural statements, ii) querying and manipulating ontologies, iii) designing rich user interfaces (UIs); b) abstraction of communication details; and c) ease of customization.

Ontologies are used in $T_{\Box}$ for data and knowledge persistence. Software modelling languages and methodologies can benefit from the integration with ontology languages in various ways, e.g., by reducing language ambiguity, enabling validation and automated consistency checking [21]. Moreover intelligent applications may be built based on ontology reasoning. In addition to providing a high level syntax with an automatic translation to platform specific code, this DSL benefits from the use of ontologies [20] which allows the user to infer knowledge, further reducing the coding effort, making $T_{\Box}$ suitable for rapid application development.

This work is part of research project, "Building decision–support through dynamic workflow systems for healthcare", a collaboration among academic researchers, a local health authority and an industry partner [12]. The goal is to develop tools for process, communication and information for community based healthcare programs. Workflow tools to guide process need to interface with a complex healthcare knowledge base and be supported by electronic forms for capturing patient information, making it accessible and usable by a variety of health providers, in a variety of service settings. While healthcare is generally governed by national or provincial standards, a change in regulations or local conditions - e.g., at the clinic, hospital or doctor's office - requires that the workflow and/or task specification easily adapt to comply with the change. Forms, in particular, are often local to a setting and may need to be modified quickly and/or frequently, to meet the needs of clinicians or administrators wishing to track some disease or specific aspect of care. Data must be stored, aggregated, interpreted and reasoned with, both locally and nationally. Ontologies are becoming the accepted method for standardizing and storing machine computable information. Healthcare is a safety critical process: technology to generate reliable software is essential.

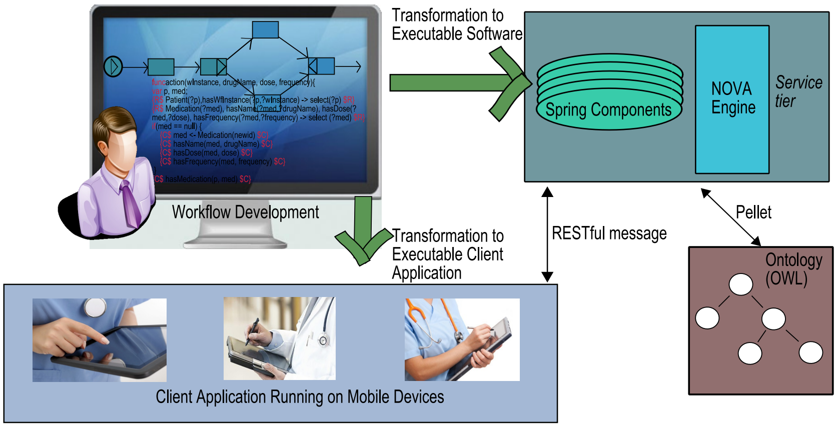
In section 2 we propose the model driven approach for workflow development; in section 3 we give details of the proposed language, $T_{\Box}$; in section 4 we include some related work, and in section 5 we conclude the paper with some directions for future work.

## 2 Model Driven Workflow Development

Workflow systems are often designed with graphical workflow modeling languages such as Business Process Modeling Notation (BPMN) [1], Yet Another

Workflow Language (YAWL) [23], Compensable Workflow Modeling Language (CWML) [16], etc. These languages use the abstract notation of process and control flow in the model to visualize the workflow; however to describe the detailed specification of a process, more is needed. Some workflow systems use XML based languages, while others use general-purpose programming languages (GPLs) such as C++, Java, etc. A control flow consists of two artefacts: i) the flow relation, and ii) branching conditions. When a workflow executes, it follows the flow relation as described in the graphical model and executes the processes (i.e., specifications written in XML or a GPL). During execution, the branching conditions are evaluated to guide the control flow.

We propose $T_\square$ for describing processes and branching conditions: $T_\square$ is a procedural language with declarative feature. For defining the flow relation and visualizing a workflow, existing workflow languages may be used. While $T_\square$ was designed so that it can be used with many workflow systems, we incorporated it into the NOVA Workflow Workbench [13]. The NOVA Workbench consist of several modules, including the NOVA Editor and the NOVA Engine. The NOVA Editor uses a graphical workflow modeling language called CWML. CWML is a block structured workflow modeling language [16] which has compensable components along with common workflow components (e.g., atomic tasks, control flow operators: XOR, OR, AND, etc.). Each atomic task in CWML is associated with a process description file (written in $T_\square$) containing procedures. XOR, OR, and Internal Choice control flow operators require that branching conditions be specified to route the flow; these branching conditions are specified in $T_\square$ as procedures which take branch numbers and return decisions.



```
funcaction(wInstance, drugName, dose, frequency){
var p, med;
{R$ Patient(?p),hasWfInstance(?p,?wInstance) -> select(?p) $R}
{R$ Medication(?med), hasName(?med,?drugName), hasDose(?
med,?dose), hasFrequency(?med,?frequency) -> select (?med) $R}
if(med == null) {
    {C$ med <- Medication(newid) $C}
    {C$ hasName(med, drugName) $C}
    {C$ hasDose(med, dose) $C}
    {C$ hasFrequency(med, frequency) $C}
}
{C$ hasMedication(p, med) $C}
```

Workflow Development
Transformation to Executable Software
Spring Components
NOVA Engine
Service tier
Transformation to Executable Client Application
RESTful message
Pellet
Ontology (OWL)
Client Application Running on Mobile Devices

**Fig. 1.** Overview of the NOVA Workflow Workbench

Fig. 1 shows the overall architecture of the NOVA Workflow Workbench, which incorporates the $T_\square$ editor, developed using Xtext. Specifications written in $T_\square$ are automatically transformed to executable Java programs, using Xtend [2]. In

NOVA Workflow, if a process description file contains a procedure named `view`, the `view` procedure is transformed to a client side Java application. All other procedures are transformed to executable Java server side programs. A `view` method may invoke other procedures; this will initiate an asynchronous data communication to the server by a Web Service (RESTful message). If a process (task) executes, meaning the execution of a procedure named either `action` or `abort` at the server side, the NOVA Workflow engine updates the control flow of the workflow according to the graphical workflow model. As the workflow executes, various operations on the ontology may be required. For example, data may be stored, removed or updated or answeres to queries may be needed. We used Pellet [19], a sound and complete OWL-DL reasoner with extensive support for reasoning. In $\mathcal{T}_\square$, ontology queries are written in the SQWRL (Semantic Query-enhanced Web Rule Language) [15] format. SQWRL is a query language for OWL [5] built on SWRL [4], a rule language which includes a high-level abstract syntax for Horn-like rules. We chose SQWRL because of its simplicity.

## 3 The $\mathcal{T}_\square$ Language

Workflow management systems often deal with many users and resources. In this section, we present a real life problem to solve with a workflow system, we introduce the $\mathcal{T}_\square$ functionality and syntax and we demonstrate the use of $\mathcal{T}_\square$ to solve the problem.

We designed a graphical workflow model from the 'Guidelines for the management of cancer-related pain in adults' [7]. The guideline suggests the use of Opioids for cancer patients. Opioids are very useful in cancer care to alleviate the severe, chronic, disabling pain but there are common side effects which include nausea and vomiting, drowsiness, itching, dry mouth, miosis, and constipation. The proper use of opioid dosage is important. Fig. 2 shows the tasks

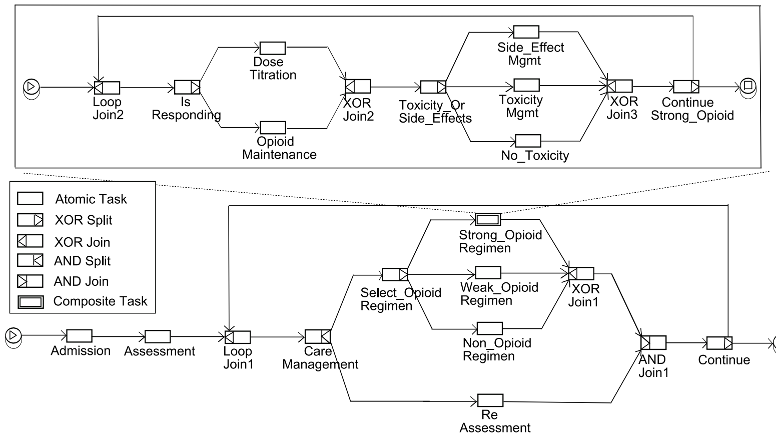


**Fig. 2.** Pain Management Workflow

for a Pain Management workflow modeled in CWML. A patient is admitted into the system and after that, pain is assessed. The 'Assessment' task assesses all causes of pain, determines pain location, pain intensity, and documents all previous analgesics. A patient is administered his prescribed medicine in one of the 'Strong_Opioid_Regimen', 'Weak_Opioid_Regimen', and 'Non_Opioid_Regimen' tasks; the 'Re_Assessment' task executes concurrently with these tasks. The 'Strong_Opioid_Regimen' is a composite task which is unfolded to a subnet workflow. This subnet workflow deals with any opioid toxicity or side effects found during the treatment procedure.

During the execution of this workflow, caregivers need to interact with the system via user interfaces. The workflow system needs to store and read information to and from a database/ontology. The workflow has decision points (e.g., Select_Opioid_Regimen determines which branch to follow) to guide the control flow. These requirements are articulated using $T_\Box$.

## 3.1 Writing Procedural Statements

Variables in $T_\Box$ are inferred variables: variable types are determined from their use. Variables in $T_\Box$ may be indexed as array indexes but a declaration of the size is not required. The size is adjusted dynamically at execution time. If no index is used, it refers to the $0^{th}$ index of a variable. In $T_\Box$, procedures may be invoked by 'call by value' or 'call by reference'. The 'call by reference' of a variable is indicated by a leading '&' . Syntax for Assignment operations, If-Else statements, For-loops, etc., in $T_\Box$ are similar to the C family languages. In $T_\Box$ every procedure returns a value, and return types are not required for procedures. In $T_\Box$ some utility procedures such as `size`, `today`, `currentTime`, `date`, `month`, `year`, `time`, and `tokenize` have been incorporated to deal with string, array, date and time data.

## 3.2 Querying and Manipulating Ontologies

In many systems, Create, Read, Update, and Delete (CRUD) operations are performed on databases. We use statements with C,R,U,D tags to perform analogous operations on an ontology Abox and refer to the relevant $T_\Box$ statements as OntAssertion, OntRead, OntUpdate, and OntDel statements.

$T_\Box$ allows us to write queries, both for task description and for branch condition, in the SQWRL format. One can perform queries combining Tbox and/or Abox syntax. The Tbox contains concepts and assertions about concepts such as subsumption ($Man \sqsubseteq Person$). The Abox contains role assertions between individuals ($hasChild(John, Mary)$) and membership assertions ($John : Man$). Similar to the 'select' operator of SQWRL, the 'select' operator in $T_\Box$ takes one or more arguments, which must be variables occuring in the body of the query. A particular value may be passed as a query criterion; if a variable is used in an ontology query without a leading question mark,?, then the value is read by the query engine. The following query retrieves all persons from an ontology with a pain intensity that is greater than 5, together with their pain intensities:

```
var  p, pain, v  =  5;
{R$ Patient(?p), hasPain(?p, ?pain), greaterThan(?pain, v) →
      select(?p, ?pain) $R}
```

The query engine will populate the variables passed as arguments of the 'select' operator. Selected results may be sorted in ascending or descending order by the 'orderBy' or 'orderByDescending' operators respectively. The following OntAssertion statements create a new 'Patient' individual and inserts a data property for the relation 'hasPain'.

```
var  p;
{C$ p  :=  Patient("Alex") $C}
{C$ hasPain(p, 6) $C}
```

Note that a reference of the newly created Patient individual is assigned to the variable 'p'. An individual may be created with an auto-incremented identity (as below) if in the ontology there exists a data property named 'hasId', where the domain of 'hasId' is 'Thing' and the range is the 'Long' data type.

```
{C$ p  :=  Patient(newid) $C}
```

OntDel statements are used to delete an individual or instance of a relation from an ontology Abox. The following code shows a delete operation of a Patient individual with id=1010.

```
var p, pid  =  1010;
{R$ Patient(?p), hasId(?p, pid) →  select(?p) $R}
{D$ Patient(p) $D}
```

In this code fragment, a search operation is performed on an ontology for a Patient individual with id=1010 and a reference is retrieved; the Patient individual's reference is then passed as an argument to the delete operation. OntUpdate statements are used to update a data property or object property of an individual. The following code fragment updates the ages of all patients whose birthday is today.

```
var p, P, bDate, Age, age, newAge, cDate  =  today();
{R$ Patient(?P), hasBirthDate(?P, ?bDate), isEqual(?bDate, cDate),
    hasAge(?P, ?Age)  →  select(?P, ?Age) $R}
foreach( p in P, age in Age){
    newAge  =  age  +  1;
    {U$ hasAge(p, age  =>  p, newAge) $U}
}
```

## 3.3 Designing User Interfaces

In many workflow applications, *forms* are used which need many UI view components such as 'Label', 'Text Field', 'Text Area', 'Check box', 'Drop down', 'Date time picker', etc., to capture user input. When a user finishes entering information in a form, the information collected is submitted to the server for processing. Sometimes client side processing on this input is required. The client side processing may involve client server communication for further information, calculation on the provided input, etc. We provide a simple syntax to develop UI forms to deal with these general requirements for client side applications.

To print text or a number in the UI, the `getLabel` procedure may be used. The `getLabel` procedure produces a 'Label' view component in the UI. One can either pass a string literal or a variable as the argument of the `getLabel` procedure. If a variable is passed to the `getLabel` procedure, then the variable is bound to a 'Label' view component. Whenever this variable is updated, the change is reflected in the 'Label'. The following code fragment produces two labels; during execution, the first label will display the text "Workflow Instance:" and the second label will display the number '112'.

```
var wid = 112;
getLabel("WorkflowInstance:");
getLabel(wid);
```

The `getText` procedure produces a 'Text Field' view component. A 'Text Field' is a common UI component to take user input. The `getText` procedure can take one or two arguments: i) a string to produce a label, and ii) a variable (optional) to display the initial text in a 'Text Field'. A destination variable name after the symbol '>>' is required for the `getText` where the user input is captured. Optionally, some statements (also known as action statements) may be written inside curly braces after the destination variable name of a `getText` procedure. These action statements will be executed when a user finishes her entry into the 'Text Field'. The following code fragment shows an example use of the `getLabel` and `getText` procedures:

```
var hospitalName, displayText = "NoInput";
getText("EnterHospitalName:") >> hospitalName{
    displayText = "Hospital:" + hospitalName;};
getLabel("EnteredText:", displayText);
```

This will produce a 'Text Field' where the user will enter text as input; the entered text will be stored in a variable named 'hospitalName'. As soon as the user finishes entering text into the 'Text Field' the action statement (enclosed with curly braces) will execute and sets a value entered by the user to the variable named 'displayText'. Since the variable 'displayText' is bound with a 'Label', when its value changes, the 'Label' view component will be updated and will display the hospital name entered in the 'Text Field'.

The `getInteger` procedure is similar to the `getText` procedure; this also produces a 'Text Field' to take input from the user; the difference is that only numbers are allowed here. The following code fragment gives an example:

```
var basicPay = 14, hourlyPay, totalHr = 0, totalSalary = 0;
getInteger("Hourlypayment : $", basicPay) >> hourlyPay {
    totalSalary = hourlyPay * totalHr; };
getInteger("TotalHourWorked : ") >> totalHr {
    totalSalary = hourlyPay * totalHr; };
getLabel("TotalSalary : $", totalSalary);
```

The first 'Text Field' will display the value of the 'basicPay' variable which is '14'. The user may change it by entering a different number; the entered number will be stored in the variable named 'hourlyPay'. The user enters the total hours worked in the second 'Text Field'. When the user finishes entering numbers in the 'Text Fields', the total salary is calculated and displayed in the UI by a 'Label'.

The `getDouble` procedure is similar to the `getInteger` procedure; the only difference is the user can enter a floating point number in the 'Text Field'. The `getDate` procedure is similar to the `getInteger` procedure but here the user enters a date in a 'Text Field' or in a 'Date Time Picker'. The `getBoolean` procedure takes one argument as input to display a title for a 'Check box' (a view component to select or de-select an item). The user may select or de-select the 'Check box' and a true or false value is assigned to the associated destination variable of a `getBoolean` procedure. If action statements are written for a `getBoolean` procedure, they will be executed after the user selects or de-selects a check box item. The following code will display a 'Check box'.

```
var painCrisis;
getBoolean("PainCrisis") >> painCrisis;
```

The destination variable 'painCrisis' is associated with the 'Check box'. If the 'Check box' is checked, a 'true' value will be set to the variable, otherwise a 'false' value will be set to the variable. Since the 'painCrisis' variable is bound to a 'Check box' view component, if the value is changed from another portion of the procedure, it will be reflected in the UI. This feature may be useful to display a UI form to update existing information. For example, if we want to display a patient's existing Pain Crisis information and allow the user to modify it, then the following code may be used:

```
var painCrisis, id = 1010;
getBoolean("PainCrisis") >> painCrisis;
{R$ Patient(?p), hasId(?p, id), hasPainCrisis(?p, ?painCrisis) →
    select(?painCrisis) $R}
```

The `getOne` procedure is used to select one item from a list of items. This procedure will either display a 'Drop down' view component (if one source variable is provided as argument) or a 'Table' with 'Radio buttons' (if more than one source variable is provided). A destination variable name is required for a `getOne` procedure. Optionally another destination variable name may be specified to store the position of the item selected from the drop down. If action statements are given for a `getOne` procedure, they will be executed as soon as the user selects an item. In the following example, a list of countries is retrieved from an ontology by performing a read operation. A `getOne` procedure is used to display the list of countries in a 'Drop down'. Another `getOne` procedure is used to display a list of provinces in another 'Drop down'. Since the provinces are different from one country to another, on the selection of a country, a further query was performed on the ontology to retrieve related province information; this was done in the action statements. The 'province' variable is bound to the source variable with the second `getOne` procedure; as a result, if a province's information was updated it will be reflected in the 'Drop down'.

```
var c, country, province, selectedProvince;
{R$ Country(?c) → select(?c) $R}
getOne("Country : ", c) >> country {
    {R$ Province(?province), hasCountry(?province, country) →
      select(?province) $R}
};
getOne("Province : ", province) >> selectedProvince;
```

Note that the 'source' variable fills a 'Drop down' view component but if we want to display one item from the items available in the 'Drop down' we may use the 'destination' variable. For instance, in a patient's admission record update form, we want to display the patient's existing province in the 'Drop down'; this can be achieved by assigning the province information to the destination variable.

The `getMultiple` procedure is similar to the `getOne` procedure but here the user may select more than one item from the source variable(s). The `getButton` procedure produces a button in the UI. When a button is pressed, the statements associated with it are executed. For example, if we want to calculate the strength of a given password then a button may be used to do this.

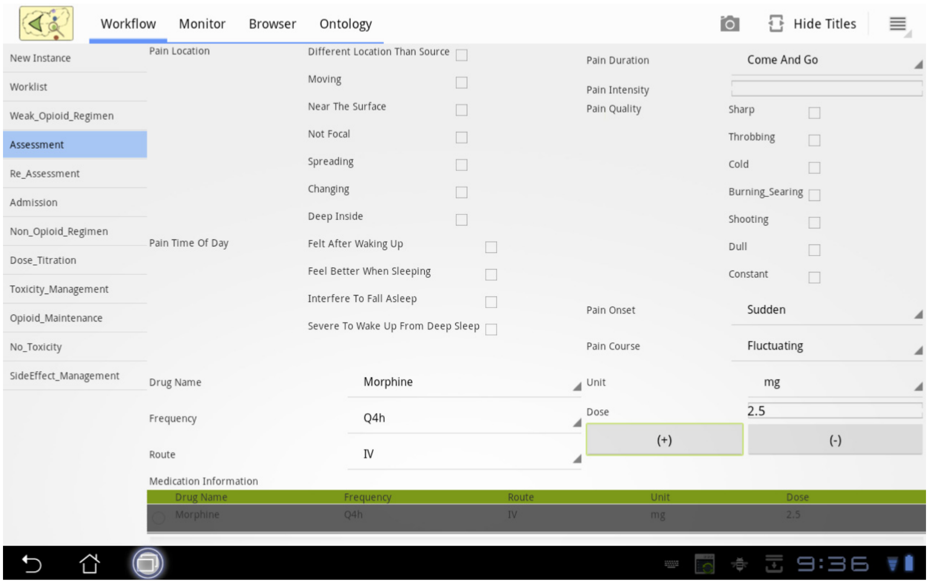
```
var password, result;
getPassword("EnterPassword") >> password;
getButton("CheckPasswordStrength") {
    result = checkPwStrength(password);
    print(result); };
```

When the button "Check Password Strength" is pressed it invokes a procedure named `checkPwStrenth` at the server with 'password' as argument and prints the result in the UI. $T_\square$ provides two procedures to arrange the view components in the UI: `openLayout` and `closeLayout`. The `openLayout` procedure takes an integer parameter which indicates the number of columns. All view components specified after an `openLayout` procedure will follow this arrangement. A `closeLayout` procedure stops putting view components in the layout format begun by an `openLayout` procedure. An `openLayout` procedure can be nested within another `openLayout` procedure allowing the user to implement a complicated table layout structure.

Now we use $T_\Box$ to design a user interface for a task from the Pain Management Workflow in Fig. 2. The code below shows the `view` procedure of the 'Assessment' task.

```
func view(){
 var wInstance, pc;
 . . .//Other variable declaration
 wInstance = getCurrentInstance(); // Get current workflow instance id
 {R$ PainCourse(?pc), hasName(?pc, ?pcName) → select(?pcName)$R}
 . . .// Read other pain assessment information from ontology
 {R$ Drug(?drug) → select(?drug)$R}
 . . .// Read Frequency, Route, Unit information from ontology
 openLayout(2); openLayout(2); // Nested Table Layout
 getMultiple("PainLocation", pLocation) >> painLocation;
 getMultiple("PainTimeOfDay", pTime) >> painTime;
 closeLayout(); openLayout(2);
 getOne("PainDuration", pDuration) >> painDuration;
 . . .// Code to display other view components
 getButton("(+)") {
 drugList[size(drugList)] = selDrug; // Adding a new drug into drugList
 . . .// Add frequency, route, dose information into list
 };
 getButton("(−)"){
 clear(drugList, medPos); // Removing selected item from list (Table)
 . . .// Remove frequency, route, dose information from list
 };
 closeLayout(); openLayout(1);
 getOne("MedicationInformation", drugList "DrugName", freqList
  "Frequency", routeList "Route", unitList "Unit", doseList "Dose")
   >> destDrug, medPos; // Showing medications in a table
 closeLayout();
 // make a submit button to send information to server
 submit(wInstance, painLocation, painTime, painDuration, . . .);}
```

$T_\Box$ generates code for client side applications using the Android [3] platform. The output of this simple 29 lines of code is shown in Fig. 3; this is a screenshot from a Tablet device (operating on an 'Android' operating system). The transformation method automatically produced 1160 lines of Java code, and a few xml configuration files to manage the Android UI, and to deal with network operations. The 'Pain Location', 'Pain Duration', 'Drug Name' etc., information comes from the ontology and is displayed in the UI. The clinician selects a drug name, frequency, route, and unit from 'Drop down' view components and inserts dose in a 'Text Field' and adds them into the 'Medication Information' table by clicking the (+) button. A medication may be removed from the table by selecting it and clicking the (−) button. A patient's opioid regimen is selected after the execution of the 'Assessment' task. We used ontology reasoning to classify medications into opioids. The rules for different opioids (e.g., strong, weak opioid) were incorporated into the ontology. In this way, complex rules of do-



New Instance
Worklist
Weak_Opioid_Regimen
Assessment
Re_Assessment
Admission
Non_Opioid_Regimen
Dose_Titration
Toxicity_Management
Opioid_Maintenance
No_Toxicity
SideEffect_Management

Pain Location
- Different Location Than Source ☐
- Moving ☐
- Near The Surface ☐
- Not Focal ☐
- Spreading ☐
- Changing ☐
- Deep Inside ☐

Pain Time Of Day
- Felt After Waking Up ☐
- Feel Better When Sleeping ☐
- Interfere To Fall Asleep ☐
- Severe To Wake Up From Deep Sleep ☐

Pain Duration: Come And Go
Pain Intensity:
Pain Quality
- Sharp ☐
- Throbbing ☐
- Cold ☐
- Burning_Searing ☐
- Shooting ☐
- Dull ☐
- Constant ☐

Pain Onset: Sudden
Pain Course: Fluctuating

Drug Name: Morphine
Frequency: Q4h
Route: IV
Unit: mg
Dose: 2.5
(+) (-)

Medication Information

| Drug Name | Frequency | Route | Unit | Dose |
|---|---|---|---|---|
| Morphine | Q4h | IV | mg | 2.5 |

**Fig. 3.** Assessment form: Output of the `view` procedure of the 'Assessment' task

main knowledge can be effectively handled by using the reasoning power of an ontology.

## 3.4 Specifying Branch Conditions

During execution of the Pain Management Workflow (Fig. 2) only one outgoing branch of the XOR task 'Select_Opioid_Regimen' is activated. A patient goes into strong opioid regimen if he is currently on a strong opioid or he is on a weak opioid with moderate, severe or unstable pain. A patient goes into the weak opioid regimen if he experiences mild but unstable pain. Otherwise he goes into the non opioid regimen. The code for the task 'Select_Opioid_Regimen' is provided below:

```
xorsplittask Select_Opioid_Regimen;
func getBranchCondition(wInstanceId, brNo){
 var p, pid, pIntensity, pUnderStrong, pUnderWeak, pc, painCourse;
 {R$ Patient(?p), hasWfInstance(?p, wInstanceId), hasId(?p, ?pid),
  hasPainIntensity(?p, ?pIntensity) → select(?p, ?pid, ?pIntensity) $R}
 {R$ PatientUnderStrongOpioid(?pUnderStrong), hasId(pid)
     → select(?pUnderStrong) $R}
 {R$ PatientUnderWeakOpioid(?pUnderWeak), hasId(pid)
     → select(?pUnderWeak) $R}
 {R$ Patient(p), hasPainCourse(p, ?pc), hasName(?pc, ?painCourse)
     → select(?painCourse) $R}
 if(brNo = 1){
  if(pUnderStrong ≠ null || (pUnderWeak ≠ null && pIntensity ≥ 4) ||
14.    (pUnderWeak ≠ null && painCourse = “GettingWorse”) ||
15.    (pIntensity ≥ 4 &&
16.       (painCourse = “Fluctuating” ||painCourse = “Getting Worse”)))
17.    return true;}
18.  else if(brNo = 2){
19.   if( (pIntensity ≥ 2 && (painCourse = “Fluctuating” ||
20.      painCourse = “GettingWorse”)) || (painCourse = “Getting Worse”))
21.   return true;}
22.  else if(brNo = 3)
23.   return true;
24. return false;}
```

The procedure `getBranchCondition` takes two parameters: a workflow instance id and a branch number and returns true for the branch that should be activated for that instance. Lines 4–11 query the ontology.

## 4 Related Work

ADEPT2 [17] is an adaptive process management system which supports dynamic change of process schema and definition. The main difference between ADEPT2 and NOVA Workflow is their underlying persistent technology and data structure; ADEPT2 does not support ontologies and the activities in ADEPT2 are written in a GPL. In ADEPT2 web forms are automatically generated from the workflow model although ADEPT2 does not allow action statements for the UI operations. ADEPT2 performs a dynamic validation of process schema change which makes the workflow system consistent. In NOVA Workflow a consistency check is performed whenever any record is inserted into or updated from an ontology Abox.

In [9], the authors presented an evolutionary approach for the model-driven construction of Web service based Web applications on the basis of workflow models which is founded on DSLs and a supporting technical framework. The Workflow DSL is an executable specification language for workflow based Web applications which allows the use of various graphical notations taken from the Business Process Modeling field, e.g., BPMN, Petri Nets, UML activity diagrams etc., as well as custom notations. This model driven approach makes development faster by reusing components but this approach is not ontology based.

In [10], the author worked on an ontology oriented programming and proposed a compiler which produces a traditional object-oriented class library that captures the declarative norms of an ontology. With this approach the developer is required to use a GPL, and the approach is not model driven. In [8], the authors described the Go! language and its use for ontology oriented programming, comparing its expressiveness with OWL. Go! allows knowledge to be represented as a set of labeled theories incrementally constructed using multiple-inheritance. This is related to our work since the authors also proposed a language for building executable ontologies. But the syntax proposed for $T_{\Box}$ is simple and abstract, and $T_{\Box}$ provides syntax for procedural statements and UI design.

In [6], Baker et. al., surveyed a large number of existing workflow systems and listed their features considering different problem aspects. None, however, follow an ontology based model driven approach as in $T_\Box$. Ontology integration and UI form generation may be specified in different ways using existing approaches but they are very easy to specify using $T_\Box$.

## 5 Conclusion

In this paper we presented a new domain specific language, called $T_\Box$, for writing specifications for tasks in workflow models. The intension of the language is reflected in the pronounciation "T-Square" – signifying speedy development of tasks (*tasks to the power of 2*). A developer may learn the simple syntax of $T_\Box$ and start developing applications without detailed knowledge of the complex API's for Ontologies, Web Services, Android platforms, etc. Code is generated automatically, allowing the developer to fully concentrate on the domain model and system analysis. If there is a change in user requirements, the developer can make the change using $T_\Box$ and the NOVA workflow system will automatically update the software accordingly. The output of NOVA Workflow is currently an Android application which runs on mobile devices but different transformations may be applied to generate other applications, for iPad, the Web, etc. End users interact with the client application.

Workflow development with $T_\Box$ can benefit from customizing existing and comprehensive ontologies, e.g., SNOMED–CT, ICNP, etc., already in use in healthcare. Since reasoning over a large ontology is time consuming, in future we will work on a bigger case study and deal with the complexity of ontology reasoning where there is a large Abox and complex rules. One approach is to use a relational database and materialize an ontology into a database [22] but this requires further research to speed up the materialization so it can support the necessary frequent updates. This will allow us to retain the benefit of using an ontology to structure and maintain complex relationships. NOVA Workflow with $T_\Box$ is currently being evaluated as part of a pilot application with our local health authority. Further $T_\Box$ functionalities are under development including a means to safeguard security of (patients') information through task-based access control, and a means to specify dynamic task scheduling.

**Acknowledgments.** We acknowledge support from the Atlantic Canada Opportunities Agencies, the Natural Sciences and Engineering Research Council of Canada, the Canadian Foundation for Innovation, and the Nova Scotia Research and Innovation Trust. We are grateful for the enthusiastic support of clinicians and administrators from the Guysborough Antigonish Strait Health Authority.

## References

1. BPMN: Business Process Model and Notation (BPMN), `http://www.omg.org/spec/BPMN/` (last accessed, January 2012)

2. Eclipse xtend, `http://www.eclipse.org/xtext/xtend/` (last accessed, January 2012)
3. Google android, `http://www.android.com/` (last accessed, January 2012)
4. SWRL, `http://www.w3.org/submission/swrl/` (last accessed, January 2012)
5. Web Ontology Language (OWL), `http://www.w3.org/2004/owl/` (last accessed, January 2012)
6. Barker, A., van Hemert, J.: Scientific Workflow: A Survey and Research Directions. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 746–753. Springer, Heidelberg (2008)
7. Broadfield, L., Banerjee, S., Jewers, H., Pollett, A., Simpson, J.: Guidelines for the Management of Cancer-Related Pain in Adults. Supportive Care Cancer Site Team, Cancer Care Nova Scotia (2005)
8. Clark, K.L., McCabe, F.G.: Ontology oriented programming in Go!. Appl. Intell. 24(3), 189–204 (2006)
9. Freudenstein, P., Nussbaumer, M., Allerding, F., Gaedke, M.: A domain-specific language for the model-driven construction of advanced web-based dialogs. In: Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, pp. 1069–1070. ACM (2008)
10. Goldman, N.M.: Ontology-Oriented Programming: Static Typing for the Inconsistent Programmer. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 850–865. Springer, Heidelberg (2003)
11. Jørgensen, J., Lassen, K., van der Aalst, W.: From task descriptions via colored petri nets towards an implementation of a new electronic patient record workflow system. International Journal on Software Tools for Technology Transfer (STTT) 10, 15–28 (2008)
12. MacCaull, W., Jewers, H., Latzel, M.: Using an interdisciplinary approach to develop a knowledge-driven careflow management system for collaborative patient-centred palliative care. In: ACM International Health Informatics Symposium, IHI 2010, Arlington, VA, USA, pp. 507–511. ACM (2010)
13. MacCaull, W., Rabbi, F.: NOVA Workflow: A Workflow Management Tool Targeting Health Services Delivery. In: The Proceedings of 1st International Symposium on Foundations of Health Information Engineering and Systems (FHIES 2011) (2011); Revised version to appear. LNCS (2012)
14. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (2005)
15. O'Connor, M.J., Das, A.K.: SQWRL: A query language for OWL. In: Proceedings of the 5th International Workshop on OWL: Experiences and Directions, OWLED 2009, vol. 529 (2009)
16. Rabbi, F., Wang, H., MacCaull, W.: Compensable WorkFlow Nets. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 122–137. Springer, Heidelberg (2010)
17. Reichert, M., Rinderle, S., Kreher, U., Acker, H., Lauer, M., Dadam, P.: ADEPT2 - next generation process management technology. In: Proceedings Fourth Heidelberg Innovation Forum, Aachen, D.punkt Verlag (April 2007)
18. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer 39(2), 25–31 (2006)
19. Sirin, E., Parsia, B., Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: Science, Services and Agents on the World Wide Web 5(2), 51–53 (2007)

20. Staab, S., Studer, R.: Handbook on Ontologies. International Handbooks on Information Systems. Springer (2004)
21. Tetlow, P., Pan, J.Z., Oberle, D., Wallace, E., Uschold, M., Kendall, E.: Ontology driven architectures and potential uses of the semantic web in systems and software engineering. History, W3C, 1–17 (2006)
22. Uddin Faruqui, R.: Scalable reasoning over large ontologies. MSc thesis, St. Francis Xavier University (expected, 2012)
23. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Information Systems 30(4), 245–275 (2005)