TMK Models to HTNs:
Translating Process Models into Hierarchical Task Networks

by

Stephen Montgomery Lee-Urban

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of science

in

Computer Science

Lehigh University

May 2005

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

_____
Date

_____
Thesis Advisor

_____
Chairperson of Department

Acknowledgements

This thesis would have been impossible without the support and guidance of Dr. Hector Muñoz-Avila; thank you!

Finally, I'd like to thank my family and friends for their unfailing encouragement and indescribable help for which all words of gratitude seem too small.

Table of Contents

List of Tables

List of Figures

Abstract

This thesis details the translation of Task-Method-Knowledge Language (TMKL) Models to Hierarchical-Task-Network (HTN) representations. It will be shown that TMKL models are as expressive as HTNs, but have a more convenient syntax, and an equivalence theorem between the two is provided. From the results of the theorem it is clear that a translation from an arbitrary TMKL model to an equivalent HTN always exists, and the particulars of a working translator constructed for this thesis by the author are explained. A synopsis of this translation schema has been accepted for publication at the upcoming Artificial Intelligence and Interactive Digital Entertainment conference.

1. Introduction

What is a process?

Loosely speaking, a process is the means by which something is accomplished via a series of actions or operations. Artificial intelligence, when seen as the attempt to make computers behave like humans, must naturally concern itself with the modeling and execution of processes. The modeling of processes is important for reasoning through concept reuse and modification; without such a model, reasoning takes place at the level of primitive actions, such as STRIPS operators (Fikes & Nilsson, 1971), and knowledge transfer to new problems is very difficult. Process models ease knowledge transfer by not only capturing the "how" of achieving a goal, but some notion of "what" is being accomplished. By reasoning at the level of "what" a more flexible overall artificial intelligence is created. One means of capturing process models is the Task-Method-Knowledge language (TMKL) process model formalism.

A simple example of these concepts involves the process of changing a light bulb (Murdock, 2001). It is a relatively simple matter to code manually a program that explicitly encodes the process in two routines, called sequentially: 1) insert the bulb, and 2) rotate the bulb. While simple and fast, this program is not flexible; there is no way for this program to respond to a request to remove the bulb since there is no model upon which the software agent can reason. An alternate approach is to abstractly represent the tasks being accomplished, the method by which tasks can be achieved, and the knowledge of what to do under situations involving decisions. The resulting encoding captures not only a plan that achieves the change bulb goal, but also models a process that is applicable to other situations.

2

Software agents using such a model would be able to reason by adapting these models of themselves in order to achieve new goals.

Why are processes important?

As illustrated by the light bulb example the representation of knowledge, including that of processes through TMKL models, yields more flexible artificial intelligence through abstraction, *meta-reasoning* on that abstraction, and *reflection* (Murdock, 2001). The gains yielded by *agent modeling* through TMKL models allows agents to automatically self-adapt to perform new tasks. This self-adaptation of the agent's reasoning processes is a transfer of knowledge from a known solution to a new problem called *analogical reasoning*. TMKL models allow for a different kind of information transfer between problems that goes beyond traditional analogical reasoning, which directly uses the results of the reasoning process. Instead, by reasoning on the level of processes, it is possible to adapt an agent in situations where the resulting plans are too different for transfer by traditional analogical reasoning (Veloso, 1994).

Many of these advantages are tightly intertwined with *machine learning*, since model-based adaptation is really a subfield of machine learning. New ways of reasoning for novel problems are produced by TMKL models since traditional machine learning techniques such as classification tasks typically restrict the problem space to selection from a fixed set of options (Dietterich, 1997). The ability to use new types of reasoning via a model is a significant contribution the machine learning community. Further, *planning* benefits from the models through their increased reuse (because of the abstraction and adaptation mechanisms previously described) and localization (on account of the TMKL models' hierarchical structuring producing natural *subtasks* which allow the segmentation of complex

3

tasks into smaller, localized subtasks/plans that can be analyzed individually by failure analysis).

The many benefits of representing an agent's capabilities through TMKL models, including what it does and how it is accomplished, can be summarized into three points: 1) Support for explanation and justification, giving an agent the means to understand its actions and justify claims on the correctness of its results, 2) Easier interoperation with other agents because a TMKL modeled agent understands its function and can inform other agents of this, and 3) Augmented learning capabilities through self understanding and modification of self-design to correct flaws and/or adapt to changes in the environment. (Murdock & Goel, 1998).

What is an HTN?

Hierarchical Task-Networks are another, older formalism used for reasoning on the abstract level of high level tasks instead of actions (Erol *et. al.*, 1994). High level tasks are successively decomposed into smaller ones until a concrete action is reached. Thus, as one moves down the decomposition hierarchy, more details of the *how* to achieve the task above become increasingly defined. The leaves of the fully expanded tree constitute the plan to be executed.

The Benefits of HTNs

Just as in TMK models, HTNs provide increased flexibility through abstraction. Unlike TMK models, however, HTNs have *clear semantics* with well defined properties and complexity analyses. One of the most important properties of HTNs is that they are proven more expressive than STRIPS-style operators (Erol *et. al.*, 1994), which is to say that all problems expressible by STRIPS are expressible by HTNs, and further there are some problems expressible in HTNs that are inexpressible in STRIPS. Another advantage of

HTNs is that they allow the encoding of strategic knowledge more naturally than other approaches (Nau *et al.*, 2000). Finally, there are a host of *implemented HTN planning systems* such as SHOP and JSHOP.

Purpose of this thesis

The purpose of this thesis is to present an equivalence proof between HTNs and TMKL models, and to describe a translation scheme between the two. To this end, a translator has been constructed. Such an equivalence shows that TMKL models have similar complexity properties as HTNs, and lends TMKL models a clear semantics which was lacking until now.

2. Task-Method-Knowledge Language (TMKL) Process Models

A Brief history of TMKL models

Before explaining what TMKL models are and how they work it is helpful to first make clear where they come from and their purpose. The main body of work presented in this thesis is based upon William Murdock's 2001 dissertation; what follows is a brief discussion on the technology's background.

The term "TMK" as a acronym for describing tasks, methods and knowledge was most notably used when explaining Interactive Kritik (Goel *et. al*., 1996, Goel & Murdock, 1996). Interactive Kritik is an extension of Kritik2 which adds a graphical user interface as a means of explaining and presenting the reasoning process of the system. The ancestor of those two systems was Kritik (Goel, 1989), a physical device design system which used Structure-Behavior-Function (SBF) models of devices. SBFs themselves were influenced by earlier approaches and largely added to the relationship between functions and behaviors. Later, Kritik2 (Goel *et. al*., 1997) extended and formalized the representation of SBF models. This extension of SBF models would later be paired with another project called Router (Goel *et. al*., 1994), which was a task-based problem solving navigational planning system that allowed tasks to be addressed by multiple methods which had selection criteria to choose amongst them. (Murdock, 2001).

The theory behind Kritik and Router was combined in Autognostic (Stroulia 1994,, Stoulia & Goel, 1995, Stroulia & Goel 1997), which takes as input a problem solver encoded in a language that would later be referred to as SBF-TMK models (Stroulia & Sorenson, 1998), and a problem to be solved. Autognostic redesigned Router and other kindred task-

based problem solvers by using a similar representation to that used for reasoning in Kritik2. In this representation, tasks are analogous to SBF functions and methods are analogous to SBF behaviors. Behind the SBF-TMK models was an approach that encoded reasoning in terms of tasks that represented often complex subsystems. This approach is different from traditional means which focus on small, atomic pieces of computation such as production rules and planning operators. The focus on tasks means that analysis of models occurs at the level of intended effects, and sequences of behaviors that achieve them adding a layer of abstraction to the reasoning process. (Murdock, 2001).

TMKL model advantages and purposes

For his dissertation Murdock developed a new, more powerful and flexible formalism for TMK models called TMKL. It is used to enable reflection in agents, which in this context means self analysis of an agent's own reasoning process and goals, by modeling the agent's composition and function through explicitly representing tasks addressed, methods applied and knowledge used. These models represent the tasks, methods and knowledge in a hierarchy, providing multiple levels of abstraction. The implementation of the TMKL modeling approach presented in Murdock's dissertation is actually an extension to a well known knowledge representation system called Loom (Brill, 1993). Knowledge in Loom is represented by concepts, instances and relations much like how knowledge is represented in semantic networks. Notable knowledge manipulation features of Loom are classification, assertion (tell), expression truth evaluation (ask), and retrieval of abstractly characterized values (retrieve). (Murdock, 2001).

TMKL models are used in the REM (Reflective Evolutionary Mind) reasoning shell developed by Murdock and Goel. The theoretical basis of REM emerges from the

observation that problems which are large and complex are often addressable by reasonably simple, specialized software. Sometimes, those specialized software systems can be modified by general-purpose mechanisms, such as reinforcement learning and generative planning, to address similar problems. When this form of adaptation is not possible, there are situations where specialized model-based adaptation strategies can accomplish parts of the adaptation process. By combining both model-based and general purpose adaptation strategies it is possible to solve problems which each method would have been unable or overwhelmingly expensive to solve individually; it is on this principle that REM operates, and TMKL derives its original purpose. It should be noted that the evolution of SBF into TMK was part of the evolution of Autognostic into the SIRRINE (Self-Improving Reflective Redesigner Including Noteworthy Experience) reasoning shell, another system developed for Murdock's dissertation research. Together, REM and SHRINE were used to implement and experiment the theories presented in his dissertation.

Among the most significant contributions of Murdock's work, also stated in chapter 1 of this thesis, are agent modeling, analogical reasoning, machine learning, planning, and meta-reasoning and reflection. The contributions to agent modeling are centered around the use of agent models for automated self-adaptation to perform new tasks. Analogical reasoning, which deals with the transfer of knowledge from old to new cases, is extended by Murdock by adapting the reasoning process themselves instead of the results of the reasoning process. Machine learning also benefits from the work since the capabilities of agents is augmented by the adaptation process and is therefore a form of machine learning. Planning benefits from TMKL models through increased reuse, since plans are abstracted into methods which are stored as models which can be used for similar problems, and localization, since

the hierarchical nature of TMKL models naturally produce sub-tasks which solve a piece of the overall process. Meta-reasoning and reflection, which is the ability for an agent to reflect on its own reasoning, are improved by TMKL models' explicit representation and separation of what is being accomplished by a process, how the process is accomplished and the knowledge used in the process. The additional knowledge about reasoning processes and its results supports additional inferences specific to model-based reasoning. The overall result of these contributions is more flexible, cost-effective reasoning systems.

TMKL Model Overview

TMKL models capture tasks (what an agent does), methods (how the agent works) and knowledge (the information used and processed by the agent). Tasks are accomplished by methods, which are in turn further decomposed into lower level tasks that are a part of the methods. A hierarchy is consequentially created where the leaves of the resulting task-method tree are primitive (not further decomposed) tasks that explicitly specify their effects, and the non-primitive tasks are the internal nodes of the tree.

TMKL Model: Task definition

Tasks encode what a piece of computation is intended to do. Table 1 depicts the components of a task (Murdock, 2001), and what kind of information required per component. The *input* and *output* components are zero or more parameters identifying the kinds of knowledge used and produced, respectively; these parameters refer to the abstract concepts instantiated by them and are defined separately (see the knowledge section). The *given* and *makes* components contain logical expressions that must be true before and after, respectively, the task is completed in order for successful execution; these expressions in Murdock's dissertation are written using Loom's formalism for queries which uses basic

logical operators, universal and existential quantification, numerical comparison, or the invocation of LISP code to combine concepts, relations, and values. Further, there are three types of tasks allowed by TMKL, broken down by the type of implementation:

1. Non primitive tasks: use the by-mmethod component to invoke any number of TMKL methods to accomplish the task. Note that the spelling convention of "by-mmethod" is an intentional naming convention adopted by Murdock.

2. Primitive tasks: use at least one of the by-procedure, assert, and binds components and have a direct representation of the effects of the task. The optional *by-procedure* component refers to a LISP procedure accomplishing the desired effect, possibly by invoking code from other programming languages. The optional *asserts* component has a logical assertion, which is a nested list of symbols in Loom's syntax for the tell operation. Finally, the optional *binds* component uses those parameters involved in the task by setting them to the values defined in the associated binds expression.

3. Unimplemented tasks: contain neither methods nor primitive information (the by-mmethod, by-procedure, asserts, and binds components are all empty) and as such must be adapted into non-primitive or primitive tasks.

Component	Type	Quantity
Input	Parameter	0+
Output	Parameter	0+
Given	Logical-expression	0-1
Makes	Logical-expression	0-1
By-mmethod	Method	0+
By-procedure	Procedure	0-1
Asserts	Logical assertion	0-1
Binds	Parameter value binding	0+

Table 1: TMKL Model Task Specification

TMKL Model: Method definition

Methods are behavioral elements that encode how a piece of computation works; the overall function of the method is encoded in the task addressed by the method. Methods essentially specify the means of accomplishing a task, and the applicability of applying a selected method of a particular task is based upon the method's provided component and the current situation. Table 2 depicts the components of a method (Murdock, 2001), and what kind of information required per component. The *provided* component is like the given component of tasks in that it specifies those facts that must be true in order to perform the method. The *additional-results* component is like the makes component of tasks in that it logically specifies the consequences of performing the method.

The *start* component of a method contains the first transition in a state-transition machine implementing the method's operation. While knowledge of the exact transition machine described by Murdock is not necessary for understanding the work presented in this thesis, an explanation is provided for completeness. It is important to note that a method can be viewed as a state machine; this becomes important when describing the translation of statements (e.g., while, for, if, etc.). These statements are elaborated in a later section of this chapter.

Each transition in the machine has a provided component like that found in a method, meaning only some of the potentially multiple outgoing transitions may be invoked in a given situation. Also, transitions link zero or more pairs of parameters used by the method to accomplish the task; an example of this type of link would be a those parameters used by a method that randomly picks a number and then displays it by calling two tasks, each with one parameter: 1) a task with an output parameter of a random number and 2) a task with an input

parameter of what to display. Finally, transitions may point to a next state in the transition machine. If no transition exists then the method is finished. (Murdock, 2001).

Component	Type	Quantity
Provided	Logical expression	0-1
Additional-result	Logical expression	0-1
Start	Transition	1

Table 2: TMKL Model Method Specification

TMKL Model: knowledge definition

Knowledge in TMKL is an explicit representation of the concepts and relations employed by an agent. Since TMKL is used by REM which in turn uses Loom for its knowledge representation, most aspects of knowledge in TMKL are from Loom. As aforementioned, knowledge in Loom is represented by concepts, instances and relations much like how knowledge is represented in semantic networks. Notable knowledge manipulation features of Loom are classification, assertion (tell), expression truth evaluation (ask), and retrieval of abstractly characterized values (retrieve) (Murdock, 2001). A complete description of Loom and its knowledge representation is outside the scope of this thesis. Further, for the purpose of understanding the translation scheme presented herein, an understanding of the knowledge component isn't strictly necessary. TMKL does add new knowledge representation constructs to Loom in order to aid in adaptive reasoning but, like the details of Looms knowledge features, an explanation of these features is not vital for understanding the translation from TMKL models to HTNs.

TMKL Model: Expression definitions

While, For, If-Then-Else, Return values, etc.

For clarity, states in the transition machine described in the "methods" section of this chapter will be termed expressions. Expressions are the same as what would be expected

from a turning computable (Turing-compete) programming language. Modern computer science theory dictates that a given programming language is Turing-complete if it is computationally equivalent to a Turing machine. That is to say that if one can provide a translation scheme from a language proven Turing-complete to the language in question then the language in question is Turing-complete. The language in question in the case of this thesis is TMKL, and the language proven Turing-complete is those implementing the HTN formalism, specifically JSHOP. JSHOP is the java implementation of SHOP (Simple Hierarchical Ordered Planner), a domain independent automated planning system. Constructs like "for," "while," "do" and "if" are all possible from the state machine, since the TMKL specification is encoded in the programming language Common Lisp and run on a modern computer. This is the underpinning idea that enables the translation from TKML to HTNs. Specifically, the three vital programming constructs exist in both TMKL and HTN: concatenation of states (sequential execution), selection between states (if/case statement), and repetition (looping via do/while/for statements). The main difference between the two formalisms reduces to "syntactic sugar," or a more convenient/alternate way of representing the same idea.

Given the existence of implemented TMKL and HTN interpreters, it seems redundant to include in this thesis a presentation of Turing machines, which typically would involve an explanation of a read-write head over an arbitrarily long length of tape recording symbols written, and capable of later reading. It suffices to say that state machines, which have preconditions on transitions and arbitrary connections between states, give rise to traditional programming language constructs and therefore those constructs fundamentally exist in both the TMKL and HTN formalisms. What is interesting is the way in which these different

representations express these constructs. Since pseudo-code has become the standard means of expression for these state machine interconnections and conditions, this thesis later presents them in that fashion.

Examples

Pseudo-code is a way of writing a program that is syntactically agnostic, or equivalently, independent of any chosen language. An example of a program written in pseudo-code is depicted in Figure 1. It should be noted that pseudo-code cannot be compiled or run on an actual Turing machine, and there are no universally accepted syntax rules. An example of what might be encoded in an arbitrary TMKL model is provided in Figure 2.

```
while further tasks remain
        select next task
        if next task has all preconditions as true
                process task
        get next task
return result
```

Figure 1: Example Pseudo-code Program

```
TMK Method Task: boolean enemyOwnsDOM( )
  If
     totalDominationPoints(td)
     totalDominationPointsOwnedbyTeam(tdTeam)
  Then
     return ( td > tdTeam )
```

Figure 2: Example TMKL Model

Concrete application: Tielt

Background on TIELT

TIELT (http://nrlsat.ittid.com), the Testbed for Integrating and Evaluating Learning Techniques, is a free software tool created to ease the evaluation of decision systems in simulators (Aha & Molineaux, 2004). The simulators can be of several different types of game genres such as real time strategy, first-person shooter, team sports games, or even a

simulator not related to gaming. One key way that TIELT makes the evaluation of decision systems easier is by reducing the number of integrations between simulators and decision systems from (m * n) to (m + n), where m is the number of investigated simulators and n is the number of decision systems being evaluated.

TSXML

TIELT decomposes the problem of decision system evaluation on performance tasks in simulators into various components, tied together via a GUI. One of these components is the Agent Description. This component provides the ability to richly define complex actions by describing them using a slightly modified TMKL model formalism. The language used to represent TIELT's TMKL model is based on XML and is called the TIELT Script Extended Markup Language (TSXML). TSXML provides a clear and uniform syntax that straightforwardly captures the TMKL model created via TIELT's interface. Because this XML based syntax might be unclear for some readers, a pseudo-code style format is used in the upcoming examples. An abbreviated example of this XML syntax is in Figure 3.

Example

```
<Method>
       <Name> pickUpStuff </Name>
       <Provided>
         <Equation>
           <FunctionReference>
             <Name> existsVisiblePickup </Name>
             <Variable>
                <Name>  </Name>
                <Type> Boolean </Type>
                <Constant> false </Constant>
             </Variable><VariableSet Name="Parameters"/>
             <IsMethod> false </IsMethod>
           </FunctionReference>
         </Equation>
       </Provided>
       <Instructions />
</Method>
```
Figure 3: Example of TIELT's TSXML Syntax

3. Hierarchical Task Networks (HTNs)

HTN overview

Before going into the specifics of Hierarchical Task-Network (HTN) planning, it is instructive to first get a general feel for it; this section contains just such a description. HTN planning is a form of planning that reasons on the level of high-level tasks instead of on the lower-level of actions (Erol *et al*., 1994). In HTN planning, high-level tasks are repeatedly decomposed into simpler ones until all tasks have been reduced into actions. This planning process is different from classic STRIPS-style planning in which planning works on the level of operators that consist of three lists of atoms: a precondition list, an add list, and a delete list (Fikes & Nilsson, 1971).

HTNs decompose high-level tasks into simpler tasks. There are three kinds of tasks: goal, primitive and compound (Erol *et al*., 1994). *Goal tasks* are like goals in STRIPS in that they are properties desired to become true in the world. *Primitive tasks* require no decomposition in that they are directly achievable by executing the corresponding concrete action. *Compound tasks*, in contrast, can be further decomposed into subtasks. Goal tasks and compound tasks are often referred to singly as non-primitive or high-level tasks. Each level in an HTN brings more details on how to achieve the high-level tasks. The sequencing of the leaves in a fully expanded HTN indicate the plan achieving the high-level tasks. In the context of game AI the decompositions can be used to encode game strategies and the leaves to actual in-game actions such as patrol, attack, etc.

The main knowledge artifacts in HTN planning are called *methods*. A method encodes how to achieve a compound task. Methods consists of 3 elements: (1) The task being

achieved, called the *head* of the method, (2) the set of preconditions indicating the conditions that must be fulfilled for the method to be applicable, and (3) the subtasks needed to achieve the head. The second knowledge artifacts are the operators. Operators in HTN planning have the same purpose as in STRIPS planning, namely, they represent action schemes. However, operators in HTN planning consist of the primitive tasks to achieve, and the effects, indicating how the world changes when the operator is applied. They have no preconditions because applicability conditions are determined in the methods.

HTN planning has two crucial advantages over STRIPS planning. Most impressively, HTN planning is provably more expressive than STRIPS planning, which is to say that there are problems that can be expressed as an HTN planning problem that cannot be expressed as a STRIPS planning problem. Further, it has been noted by several authors that encoding strategic knowledge is more natural in HTNs than in STRIPS. Even though it is generally possible to encode strategies in STRIPS representations, HTNs capture strategies naturally because of the explicit representation of stratified interrelations between tasks. Furthermore, representing HTNs in STRIPS operators is very cumbersome in general (Lotem & Nau, 2000) and sometimes even impossible (Erol *et al.*, 1994).

Formal presentation of HTN planning

Having presented the basics of HTN planning, it is now appropriate to give a formal definition. An arbitrary language L used for HTN planning represents the world and actions similarly to STRIPS-style planning. The vocabulary of language L is a tuple $<V, C, P, F, T, N>$, defined as follows:

- $V = \{v_1, v_2, \ldots\}$, an infinite set of variable symbols

- C = a finite set of constant symbols representing objects

- P = a finite set of predicate symbols representing relations among the objects

- F = a finite set of primitive task symbols representing actions

- T = a finite set of compound task symbols

- $N = \{n_1, n_2, \ldots\}$, an infinite set of symbols used for labeling tasks

The three types of tasks enumerated in the previous section can be explicitly defined using the above vocabulary. Specifically, if $x_1, \ldots, x_k$ are terms, then a primitive task has the form $do(f(x_1, \ldots, x_k))$, where f is an element of F. A goal task has the form $achieve(l)$, where l is a literal. Finally, a compound task has the form $perform[t(x_1, \ldots, x_k)]$, where t is an element of T (Erol *et al.*, 1994). HTN planning connects tasks and constraints on those tasks together into a task network. These task networks abstractly take the form $((n_1 : \alpha_1), \ldots, (n_m : \alpha_m), \phi)$, where α_i is a task labeled with n_i, and ϕ is a boolean formula which optionally contains negation and disjunction and consists of variable binding constraints (e.g., $v = v'$, $v = c$), temporal ordering constraints (e.g., $n < n'$), and truth constraints (e.g., (n,l), (l,n), and (n,l,n')) (Erol *et al.*, 1994). An example task network and its associated graphical representation is shown in Figure 4, from (Erol *et al.*, 1994). Depicted are three tasks and associated constraints. The constraints include that v_1 should be moved last, v_1 and v_2 are to remain clear until v_1 is moved, and that v_3 is bound to the location of v_1 before v_1 is moved.

$$((n_1 : achieve[clear(v_1)])(n_2 : achieve[clear(v_2)])$$
$$(n_3 : do[move(v_1, v_3, v_2)])$$
$$(n_1 \prec n_3) \wedge (n_2 \prec n_3) \wedge (n_1, clear(v_1), n_3)$$
$$\wedge (n_2, clear(v_2), n_3) \wedge (on(v_1, v_3), n_3)$$
$$\wedge \neg(v_1 = v_2) \wedge \neg(v_1 = v_3) \wedge \neg(v_2 = v_3))$$

n_1:

achieve[clear(v_1)] — $clear(v_1)$

n_3:

:on(v_1, v_3) **do[move(v_1, v_3, v_2)]**

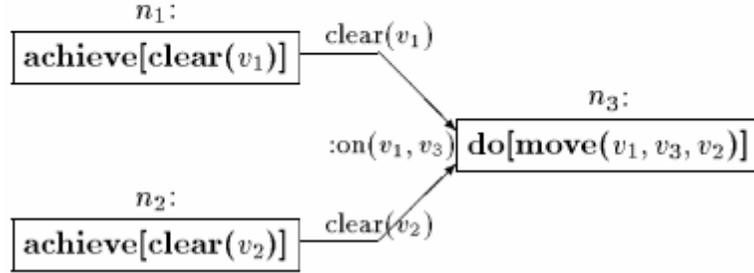n_2:

achieve[clear(v_2)] — $clear(v_2)$

Figure 4: Example Task Network from (Erol *et al.*, 1994)

Operators, as described in the last section, associate effects to primitive task symbols and specify how actions change the world. They are of the form $(f(v_1, \ldots, v_k), l_1, \ldots, l_m)$ where f is a primitive task symbol, $v_1, \ldots, v_k$ are variable symbols, and $l_1, \ldots, l_m$ are postconditions, which are literals denoting the primitive task's effects. Operators have no preconditions, and are achieved by executing the corresponding action. A plan is therefore a sequence of ground primitive tasks, and is appropriate for a given *initial state* of the world which is also a list of ground atoms. Primitive tasks are those that can be performed directly, whereas non-primitive are those requiring further planning or reasoning to determine how to perform them.

Methods are pairs of the form (α, d), where α is a non-primitive task and d is a task network like that shown in Figure 4. The pair encodes the knowledge that one way the task α can be achieved is by achieving the task network d. A task network is achieved when all its subtasks are performed without violating the constraint formula ϕ.

Finally, a planning domain is expressed as a pair $D = \langle Op, Me \rangle$ where Op is a set of operators and Me is a set of methods. A planning problem is expressed as a triple $\mathbf{P} = \langle d, I, D \rangle$, where D is a planning domain, I is the initial state, and d is the task network in which a plan is desired; $\mathbf{P}$ is termed *primitive* if d contains only primitive tasks, termed *propositional* if no variables are allowed, and termed *totally ordered* if all the tasks in any task network are totally ordered. HTN planning starts with an initial task network d and performs the following steps:

1) find a non-primitive task u in d and a method $m = (t, d')$ in M such that t unifies with u.

2) modify d by replacing u with the tasks in d' and incorporate the constraints of d' into d

3) repeat steps 1 and 2 until no non-primitive tasks are left in d

4) find a totally-ordered ground instantiation σ of d that satisfies all constraints in ϕ. If such a totally-ordered ground instantiation satisfying ϕ is found then σ is a successful plan for the original problem.

Soundness and Completeness of the HTN planning algorithm

Two properties of a planning algorithm are particularly important to AI researchers. The first property is soundness, which asks whether every plan returned from the algorithm really works (the plans are valid solutions to the original problem); the second property is completeness, which says that given a solvable problem, the planner finds a solution, which is not the same as saying that every plan will be found (Weld, 1994). In (Erol *et al.*, 1994b) it is proven that the HTN planning algorithm is both sound and complete.

Expressiveness of HTNs

Expressiveness, which is a property concerning the types and complexity of problems that can be encoded by a particular formalism, is another property explored by Erol *et al*. (1994). It is shown that the HTN formalism presented in that paper can express situations impossible to express using traditional STRIPS operators. Further, any problem that can be expressed by STRIPS operators can be expressed as an equivalent HTN by transforming every STRIPS operator into a HTN primitive task symbol and transforming every precondition and effect of STRIPS operators into an HTN method whose preconditions are the same and which calls the appropriately transformed primitive task symbol. Such a transformation means that STRIPS planning is actually a special case of HTN planning. This relationship is likened to the relationship between context-free languages and regular languages in that compound tasks represent sets of primitive task networks just as non-terminal symbols represent sets of sets of terminal symbols.

HTN planning is NP-complete

Recall the definitions of primitive, totally ordered, and propositional plans from the section in this thesis on the formal representation of HTN planning. It is shown in Erol *et al*. (1994) that if a planning problem is primitive and either/neither propositional or/nor totally ordered then the plan existence problem is NP-complete. This arises from the fact that under these assumptions it is simple to nondeterministically guess a total ordering and variable binding on the task network's constraint formula ϕ, and verify that ϕ is satisfied in polynomial time. Under the same assumptions, the constraint language expressing ϕ can represent the satisfiability problem, thus creating NP-hardness.

4. The Translator

TMKL models at first appear to be more expressive than HTNs since the language of the former explicitly provides constructs for looping, conditional execution, assignment, functions with return values, and other features not found in HTNs. However the research supporting this thesis shows that HTNs implicitly provide support for the same features, albeit in a less obvious fashion, and a translation from TMKL models to HTNs is always possible. For the sake of clarity, pseudo-code is used for describing the HTNs instead of the LISP-based syntax used in HTN planners like JSHOP.

Summary table

The following table illustrates in plain language the mapping between those TMK Model constructs that do not exist in HTNs and the original HTN formalism. Later sections will further elaborate upon the specifics of each translation. The intent of Table 3 is to give an intuitive feel for the equivalency.

TMK Models	HTNs
Tasks have preconditions	Add preconditions to methods
If-then-else	Use HTN method syntax
Call	Subtask
Set (Assignment)	Split into new method and pass in evaluated value
Return (values from functions)	Use unbound variable as parameter in caller's invocation; set same variable in callee's preconditions
For (iteration)	Change to while, recursion
While (iteration)	Recursion

Table 3: Plain Language Mapping from TMK Models to HTNs

Brief Description

The formal proof: generic TMKL model into HTN

The equivalence begins with the top level definition of the translation function, TRANS, which takes as input a TMKL model file and outputs an equivalent HTN (Figure 5): TRANS(TMKL Model) $\Rightarrow$ HTN. Then, for each language element in of TMKL (represented in pseudo-code), a semantically equivalent HTN statement or series of statements is provided using a recursive definition of the TRANS function. Following each pseudo-code translation is a brief commentary on interesting aspects of the depicted translation. Note 1) that the purpose of the translations is to show equivalence as clearly as possible, and as such the efficiency of the pseudo-code is sometimes sacrificed, and 2) the pseudo-code presented fully defines all valid TMKL inputs to the TRANS function, therefore producing a complete HTN translation.

The following are conventions used in presenting the pseudo-code in this section:

- An HTN method has both an imperative and declarative form. The declarative form is the fully defined HTN method whereas the imperative form is the form used when calling the method as a sub-task (and is therefore just the method head with appropriate parameters).

- The Backus-Naur "*<type>*" notation is loosely used, but the explanations following each translation disambiguate what is meant.

- The pseudo-code at times refers to objects before they are declared (e.g., using the imperative form of method *m* before it is created). This is for clarity of

presentation, and in every case the object in question is created within the "behavior" section of the containing TRANS function.

- A TMKL model statement σ_i is of the form "while <expression> do <body>". Sets of statements, as found in elements like <body>, are treated recursively; this means that statement numbering, which effects that which is referred to as the next TMKL statement, has a *scope* relative to the encapsulating body. Thus, the statement following σ_i is still naturally referred to as statement σ_{i+1} and the translation of the <body> has its own localized sequential order. Further, every set of TMKL model statements creates a scope that has a set of variables used; this set is passed to statement sets nested within the original statement set.

- HTNMethodSet has a ".imperativeForm" which is the imperative form ψ of an HTN task or HTN method. A HTNMethodSet is an ordered set of HTN methods created via "Create HTN method *m*" in the "behavior" section of a TRANS function.

- HTNMethodSet has a ".variables" which is an ordered set of all HTN variables used in the top-level scope of the originating set of TMKL model statements

Top level recursive function

function: TRANS
input:
   Ordered set of TMKL statements τ
   (optional) the imperative form ψ of an HTN task or HTN method
output: Equivalent HTN η, returned as a set of HTN tasks and HTN methods
behavior:
   Declare result set δ

   For each top-level TMKL statement σ_i, in τ do
    Declare HTNMethodSet ρ

    If not last nor terminal (return) statement in τ then
     Set ρ = TRANS(σ_i, TRANS(σ_{i+1}, null).imperativeForm)
    Else
     Set ρ = TRANS(σ_i, ψ)
    End If

    add each element of ρ to result set δ
   End For

   If first element of τ (TMKL statement σ_1) is not empty then
    Set δ.imperativeForm = TRANS(σ_1, null).imperativeForm
   Else
    Set δ.imperativeForm = ψ
   End if

   Return δ
   //To print, iterate through set and print declarative form!
   //invoke η.imperativeForm from top level task to start HTN planning

Figure 5: Top Level Recursive Function

The above is the top level recursive translation function. All further definitions overload this function until a complete language mapping is created (all statements have translations). The most interesting part of this function is the part dealing with the last/terminal statement in τ. By checking this condition and acting upon it as depicted, the interconnection of TMKL model statement sequences is created. Often, the translation of a TMKL model statement σ_i results in multiple HTN methods. One (or many) of the resultant

26

HTN methods must invoke the imperative form of the translated TMKL model statement σ_{i+1}. Recall that TMKL statement sets maintain their own ordering relative the scope they create. To this end, the input parameter ψ allows for the next imperative HTN method invocation to come from a statement or series of statements belonging to a larger scope.

If-Then-Else

function: TRANS
input:
        TMKL statement σ_i = if <cond> then <body1> else <body2>
            <cond> is a TMKL expression
            <body1> is an ordered set of TMKL statements
            <body2> is an ordered set of TMKL statements
        the imperative form ψ of the translated TMKL statement σ_{i+1}
output: Equivalent HTN η, returned as a set of HTN tasks and HTN methods
behavior:
            Declare result set δ
            Declare HTNMethodSet ρ'
            Declare HTNMethodSet ρ''

            Set ρ' = TRANS(<body1>, ψ)
            Set ρ'' = TRANS(<body2>, ψ)

            Create HTN method m:
              name = m
              parameters = all parameters of same/higher scope in TMKL model
              precondition set 1 = TRANS(<cond>)
              sub-task set 1 = m'.imperativeForm
              precondition set 2 = true
               sub-task set 2 = m''.imperativeForm
            add m to result set δ

            Create HTN method m':
               name = m'
               parameters = all parameters of same/higher scope in TMKL model
              precondition set 1 = true
              sub-task set 1 = ρ'.imperativeForm
            add m' to result set δ

            Create HTN method m'':
               name = m''
               parameters = all parameters of same/higher scope in TMKL model
              precondition set 1 = true
              sub-task set 1 = ρ''.imperativeForm
            add m'' to result set δ

            Set δ.imperativeForm = m.imperativeForm

            Return δ

Figure 6: If-Then-Else Translation Function

The If-Then-Else behavior is implemented in HTN method m, where the HTN syntax is directly exploited. The HTN precondition/sub-task set pairs have their preconditions evaluated sequentially (1..n) until the first positive evaluation in pair i, at which point the sub-task set i is expanded into the further translated methods. Thus, a translation of <cond>, defined in a later section, is used as the preconditions on the precondition/sub-task set 1 pair of HTN method m; if this translated condition is evaluated as true when performing HTN planning, then the associated HTN method m' is invoked. Otherwise, the precondition/sub-task set 2 is used to create the else, and since its preconditions are always met, the HTN method m'' is invoked. Note that the recursive calls to the top-level TRANS function for ρ' and ρ'' ensures that:

1. The HTN methods referred to in m' and m'' are implemented as the translated TMKL statements of <body1> and <body2>

2. The last/terminal (e.g. "return") statement in <body1> and <body2> is the imperative form ψ of the translated TMKL statement σ_{i+1} that follows the 'If-then-else' (thereby creating sequential execution of statements).

3. The interconnection of TMKL model statement sequences is created is maintained within the recursive translation of <body1> and <body2> by a "local" ψ being created in the recursive TRANS. For the sake of simplicity, hereafter these interconnected sequences are not highlighted, but they are an integral part of the translation.

Call

```
function: TRANS
input:
        TMKL statement σ_i = call <functionCall>
                <functionCall> is a call to a TMKL function
        the imperative form ψ of the translated TMKL statement σ_{i+1}
output: Equivalent HTN η, returned as a set of HTN tasks and HTN methods
behavior:
        Declare result set δ

        Create HTN method m:
          name = m
          parameters = all parameters of same/higher scope in TMKL model
          precondition set 1 =
          sub-task set 1 = m'.imperativeForm
        add m to result set δ

        Create HTN method m':
          name = m'
          parameters = all parameters of same/higher scope in TMKL model
          precondition set 1 =  TRANS( <functionCall> )
          sub-task set 1 = ψ
        add m' to result set δ

        Set δ.imperativeForm = m.imperativeForm

        Return δ
```

Figure 7: Call Translation Function

The call is essentially a pass-through translation from a TMKL function to an equivalently named HTN method. Recall that both the TMKL described in this thesis and the HTN planner JSHOP allow invocation of external LISP functions. Translation is therefore naturally equivalent, and is completed by including as the next sub-task a HTN method call to the next translated statement or body. Two HTN methods must be created in this case so that the translation of <functionName> can modify variables, and have these changes reflected in the next sub task. For further discussion regarding how variables are handled, see the section on the "set" and "function calls" statements.

30

Set (assignment)

```
function: TRANS
input:
        TMKL statement σ_i = set <variable> to <expression>
                <variable> is a TMKL variable
                <expression> is a TMKL expression
        the imperative form ψ of the translated TMKL statement σ_{i+1}
output: Equivalent HTN η, returned as a set of HTN tasks and HTN methods
behavior:
                Declare result set δ

                Create HTN method m:
                    name = m
                    parameters = all parameters of same/higher scope in TMKL model
                    precondition set 1 =
                    sub-task set 1 = m'.imperativeForm
                add m to result set δ

                Create HTN method m':
                    name = m'
                    parameters = all parameters of same/higher scope in TMKL model
                    precondition set 1 =
                        TRANS( <variable> ) =  TRANS( <expression> )
                    sub-task set 1 = ψ
                add m to result set δ

                Set δ.imperativeForm = m.imperativeForm

                Return δ
```

Figure 8: Set Translation Function

The set statement is essentially a two part TRANS call that assigns the translated expression to the translated variable in the precondition of an HTN method. In order to enable the updating of variable values, JSHOP requires that changes be made in the preconditions section of a method. Note that in the imperative form of ψ the updated variable name must be used, and that the <expression> can also be a <function> (see the section on translating function calls). If the <expression> is of non-boolean type then a direct assignment cannot be used; instead a parameter is added to the <expression>

31

translation and it is set in the precondition of the final HTN method implementing the translation.

Return values from functions

> **function:** TRANS
> **input:**
>   TMKL statement σ_i = <expression>(<parameters>)
>   <expression> is a TMKL expression
>   <parameters> is a set of TMKL variables
>   the imperative form ψ of the translated TMKL statement σ_{i+1}
> **output:** Equivalent HTN η, returned as a set of HTN tasks and HTN methods
> **behavior:**
>   Declare result set δ
>
>   Create HTN method m:
>   name = m
>   parameters = all parameters of same/higher scope in TMKL model
>   precondition set 1 =
>   TRANS(<expression>)
>   sub-task set 1 = ψ
>   add m to result set δ
>
>   Set δ.imperativeForm = m.imperativeForm
>
>   Return δ

Figure 9: Return Translation Function

Return values from functions are handled like "set" statements in that an additional parameter is added to the <expression> translation; the additional parameter is used to return the result. This result parameter is set in the precondition of the final HTN method implementing the translation.

For (iteration)

---

function: TRANS
input: TMKL statement σ$_i$ = for(<init>; <cond>; <incr>) <body>
                <init> is a TMKL expression "set <variable> to <expression>"
                <cond> is a TMKL expression
                <incr> is a TMKL expression "set <variable> to <expression>"
                <body> is an ordered set of TMKL statements
      the imperative form ψ of the translated TMKL statement σ$_{i+1}$
output: Equivalent HTN η, returned as a set of HTN tasks and HTN methods
behavior:

        Declare result set δ
        Declare HTNMethodSet ρ
        Set ρ = TRANS(<body>, null)
        add each element of ρ to result set δ

        Create HTN method m:
          name = m
          parameters = all parameters of same/higher scope in TMKL model
          precondition set 1 = TRANS(<init>)
          sub-task set 1 = m'.imperativeForm
        add m to result set δ
        Create HTN method m':
          name = m'
          parameters = all parameters of same/higher scope in TMKL model
          precondition set 1 = TRANS(<cond>)
          sub-task set 1 = m''.imperativeForm
          precondition set 2 = true
          sub-task set 2 = ψ
        add m' to result set δ
        Create HTN method m'':
          name = m''
          parameters = all parameters of same/higher scope in TMKL model
          precondition set 1 = ρ.imperativeForm
          sub-task set 1 = m'''.imperativeForm
        add m'' to result set δ
        Create HTN method m''':
          name = m'''
          parameters = all parameters of same/higher scope in TMKL model
          precondition set 1 = TRANS(<incr>)
          sub-task set 1 = m'.imperativeForm
        add m''' to result set δ
        Set δ.imperativeForm = m.imperativeForm
        Return δ

Figure 10: For Translation Function

The HTN method m ensures that the iterating variable is appropriately initialized before calling method m', which evaluates the translated condition and invokes the translation of the "for" body when true, and invokes ψ otherwise. An interesting feature to note is that method m'' creates the HTN translation of the body of the for by invoking the body's imperative form in the precondition (allowing variables to be updated). Since the preconditions of m'' always evaluate to true, method m'' always calls m'''.imperativeForm. Method m''' handles the incrementation of the iterated variable, and is actually equivalent to a "set" translation. It should be noted that since m''' will already have as a parameter the incremented variable, a uniquely named variable must be used for the assignment of the result of the incrementation calculation. This unique variable will then be used in m'.imperativeForm by m''', thereby creating the loop effect with an updated variable (recall that variables in JSHOP can only take on a value once, and this essentially un-defines and re-defines the incrementing variable).

While (iteration)

---

function: TRANS
input:
       TMKL statement σ_i = while <cond> do <body>
          <cond> is a TMKL expression
          <body> is an ordered set of TMKL statements
      the imperative form ψ of the translated TMKL statement σ_{i+1}
output: Equivalent HTN η, returned as a set of HTN tasks and HTN methods
behavior:
         Declare result set δ
         Declare HTNMethodSet ρ

         Set ρ = TRANS(<body>, null)
         add each element of ρ to result set δ

         Create HTN method m:
           name = m
           parameters = all parameters of same/higher scope in TMKL model
           precondition set 1 = TRANS(<cond>)
           sub-task set 1 = m'.imperativeForm
           precondition set 2 = true
           sub-task set 2 = ψ
         add m to result set δ

         Create HTN method m':
           name = m'
           parameters = all parameters of same/higher scope in TMKL model
           precondition set 1 = ρ.imperativeForm
           sub-task set 1 = m.imperativeForm
         add m' to result set δ

         Set δ.imperativeForm = m.imperativeForm

         Return δ

Figure 11: While Translation Function

The while is a much simpler implementation of the same logic used in translating the

"for" statement. Method m evaluates the translated condition and invokes the translation of

the while body when true, and invokes ψ otherwise. An interesting feature to note is that, as

done in the "for" translation, method m' creates the HTN translation of the body of the while

by invoking the body's imperative form in the precondition (once again, allowing variables to be updated). Since the preconditions of m' always evaluate to true, method m' always makes the recursive invocation m.imperativeForm.

Function Calls

```
function: TRANS
input:
        TMKL function call func = <functionCall>
output: An equivalent HTN function call func'
behavior:
            If func is a standard LISP function call then
                Return <functionCall + returnVariable>
            End if
```

Figure 12: Translation of Function Calls

This form of the TRANS function is directly used by the "call" statement. As aforementioned, return values from functions are handled like "set" statements in that an additional parameter is added to the *func* translation; the additional parameter is used to return the result in the case that the function is also implemented as a TMKL process model. Otherwise, if *func* is a LISP function then the TMKL models and HTNs are literally equivalent (use the same invocation).

Variables

```
function: TRANS
input:
        TMKL variable var = <variable>
output: An equivalent HTN variable var'
behavior:
            Return <variable>
```

Figure 13: Variable Translation Case 1

```
function: TRANS
input:
        Set of TMKL variables varSet = <variables>
output: An equivalent set of HTN variables varSet'
behavior:
            Declare ordered result set res

            For each TMKL model variable var in varSet do
               Add TRANS( var ) to res
            End for

            Return res
```

Figure 14: Variable Translation Case 2

These TRANS functions, Figure 13 and Figure 14, ensure that the variable names

used in the HTN are valid, since it is possible that a certain TMKL model will allow variable

names that a particular HTN will not. The particulars of how variables are handled for a

given TRANS are detailed in the other subsections in this chapter.

Expressions

```
function: TRANS
input:
        TMKL expression exp = <expression>
output: An equivalent HTN expression
behavior:
            Declare HTN expression ans

            If expression exp is a TMKL function call
               Declare TMKL function call funcCall
               Set funcCall = exp
               Set ans = TRANS( funcCall )
            Else
               Set ans = exp
            End If

            Return ans
```

Figure 15: Translation of Expressions

If *exp* is a TMKL function call then the transformation invokes the "function calls"

TRANs function. Otherwise *exp* is taken to mean a standard constraint formula containing

conjunctions, disjunctions and negations. Both TMKL models and HTNs use the same formalism for constraint formulas and as such the translation is literally equivalent (use the same invocation).

Equivalence Theorem

To date it has been unclear in the literature whether TMKL model planning is more expressive than HTN planning. According to Erol *et*. *al*. (1994b) there is not a well established definition of expressivity for planning languages; expressivity has been defined based on model-theoretic semantics, operational semantics, and on the computational complexity of problems representable by the planning language (Erol *et al*., 1995).

A presentation of equivalence based upon the definition of model-theoretic expressivity described in (Erol, 1994b) would explicitly make clear the equivalence of TMKL models and HTNs. However, doing so requires introducing and explaining new concepts; thankfully there is a simpler way to set about this proof. The equivalence presented in this thesis is based upon Erol *et*. *al*. (1995) in which it states that the expressivity of two languages can be compared by demonstrating that a polynomial or Turing computable transformation exists. It is precisely this demonstration that has been provided in the section titled "The formal proof: generic TMKL model into HTN". Thus:

> **Theorem:** There exists a Turing-computable function ψ from the set of TMKL model planning problem instances to the set of HTN planning problem instances such that for any TKML model planning problem instance **P**, and any plan σ, σ solves **P** iff $\psi(\sigma)$ solves $\psi(\mathbf{P})$.

Both **P** and σ as used in the theorem are defined in Chapter 3 for HTNs; given the existence of a computable transformation (as shown by the function TRANS in this section

and in the existence of a working translator built for this thesis), both $\mathbf{P}$ and σ can apply to

TMKL models and therefore lend their well defined semantics to TMKL models.

5. Conclusion

Summary

This thesis has presented an equivalence proof between HTNs and TMKL models by presenting a Turing-computable function that maps the set of TMKL model constructs to equivalent HTN constructs. This function was illustrated via a set of translation functions, implemented in pseudo-code. Further, a working translator has been constructed to translate TSXML to JSHOP's HTN planning language formalism. Such an equivalency shows that TMKL models have similar complexity properties as HTNs, lends TMKL models a clear semantics that the literature has lacked, shows that a translation from an arbitrary TMKL model to an equivalent HTN always exists, and has the implications detailed in the next section. A synopsis of this translation schema has been accepted for publication at the upcoming Artificial Intelligence and Interactive Digital Entertainment conference.

Implications

TMKL Model Plans are Sound and Complete

Chapter 2 described that the HTN planning algorithm is sound and complete. Since TMKL models are equivalent to HTNs, TMKL model planning is also sound and complete.

TMKL Models are more expressive than STRIPS

Chapter 2 described that HTNs are strictly more expressive than STRIPS. Since TMKL models are equivalent to HTNs, TMKL models are also strictly more expressive than STRIPS.

TMKL Model planning is NP-Complete

Chapter 2 described that HTN planning is NP-Complete under certain restrictions. Under these same restrictions, and given that TMKL models are equivalent to HTNs, TMKL models planning is also NP-Complete.

Future Work

One next step to take in extending the work presented in this thesis would be to test the translator by first translating a TIELT agent description, then creating a plan from JSHOP, and finally executing the plan in an experiment running in TIELT. Further, the translator could be extended to handle objects/structures instead of just the basic types (i.e., Boolean, Real, etc.). Another next step is to prove equivalence via other methods such as model-theoretic semantics, operational semantics.

Bibliography

Aha, D.W., & Molineaux, M. (2004). *Integrating learning in interactive gaming simulators*. Challenges of Game AI: AAAI'04 Workshop Proceedings (Technical Report WS-04-04). San Jose, CA: AAAI Press.

Brill, D. (1993). Loom reference manual. http://www.isi.edu/isd/LOOM/documentation/manual/quickguide.html. Accessed May 2005.

Dietterich, T.G. (1997). *Machine learning research: Four current directions*. AI Magazine, 18(4): 97-136.

Erol, K., Hendler, J., & Nau, D. (1994). *HTN planning: Complexity and Expressivity*. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI94).

Erol, K.; Hendler, J.; & Nau, D. (1994b). *Semantics for Hierarchical Task Network Planning*. Technical report CS-TR-3239, UMIACS-TR-94-31, ISR-TR-95-9, Computer Science Dept., University of Maryland, March 1994.

Erol, K., Hendler, J., & Nau, D. (1995). *Complexity results for HTN planning*. Annals of Mathematics and Artificial Intelligence

Fikes, R., & Nilsson, N. (1971). *Strips: a new approach to the application of theorem proving*. AI.

Goel, A. K. (1989). *Integration of Case-Based Reasoning and Model-Based Reasoning for Adaptive Design Problem Solving*. PhD dissertation, The Ohio State University, Dept. of Computer and Information Science.

Goel, A. K., Ali, K., Donnellan, M., Gomez, A., & Callantine, T. (1994). *Multistrategy adaptive navigational path planning*. IEEE Expert, 9(6):57-65.

Goel, A. K., Gomez, A., Grué, N., Murdock, J. W., Recker, M., and Govindaraj, T. (1996). *Explanatory interface in interactive design environments*. In Gero, J. S. and Sudweeks, F., editors. Proceedings of the Fourth International Conference on Artificial Intelligence in Design - AID-94, pages 387-405, Stanford, California. Kluwer Academic Publishers.

Goel, A. K., & Murdock, J. W. (1996). *Meta-cases: Explaining case-based reasoning*. In Smith, I. and Faltings, B., editors, Proceedings of the Third European Workshop on Case-Based Reasoning - EWCBR-96, Lausanne, Switzerland. Springer.

Goel, A. K., Bhatta, S. R., & Stroulia, E. (1997). *Kritik: An early case-based design system*. In Maher, M. L. and Pu, P., editors, Issues and Applications of Case-Based Reasoning to Design. Lawrence Erlbaum Associates.

Lotem, A., & Nau, D. S. (2000). *New advances in GraphHTN: Identifying independent subproblems in large HTN domains*. AIPS-2000 Proceedings, AAAI Press.

Murdock, J. W., & Goel, A. K. (1998). *A functional modeling architecture for reflective agents*. In Proc. AAAI98 Workshop on Functional Modeling and Teleological Reasoning, Madison, Wisconsin.

Murdock, J. W. (2001). *Self-Improvement through Self-Understanding: Model-Based Reflection for Agent Adaptation*. Dissertation for Georgia Institute of Technology.

Nau, D. S., Aha, D. W., & Muñoz-Avila, H. (2000). Ordered task decomposition. AAAI-2000 Workshop on Representational Issues for Real-World Planning Systems. AAAI Press.

Stroulia, E. (1994). *Failure-Driven Learning as Model-Based Self Redesign*. PhD dissertation, Georgia Institute of Technology, College of Computing.

Stroulia, E. & Goel, A. K. (1995). *Functional representation and reasoning in reflective systems*. Journal of Applied Intelligence, 9(1):101-124. Special Issue on Functional Reasoning.

Stroulia, E. & Goel, A. K. (1997). *Redesigning a problem-solver's operators to improve solution quality*. In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI) 97, pages 562-567, San Francisco. Morgan Kaufmann Publishers.

Stroulia, E. & Sorenson, P. (1998). *Functional modeling meets meta-CASE tools for software evolution*. In Proceedings of the International Workshop Software Program Evolution.

Veloso, M. (1994). Planning and learning by analogical reasoning. Springer-Verlag, December 1994.

Weld, D. (1994). An Introduction to Least-Commitment Planning. AI Magazine, 15(4): 27-6

Vita

Stephen Montgomery Lee-Urban was born in Long Island, NY to Kathryn and James Urban. He attended Lehigh University from the fall of 1999 through the spring of 2005, receiving a B.S. in Computer Engineering in 2003 with highest honors, and his Masters (pending) in 2005. He has one publication in the Artificial Intelligence and Interactive Digital Entertainment (AIIDE05) conference.