Semi-Formal Functional Software Modeling with TMK

J. William Murdock
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332, USA
murdock@cc.gatech.edu

February 11, 2000

Technical Report GIT-CC-00-05

Abstract

The Task-Method-Knowledge (TMK) modeling language is a formal mechanism for describing a software system. It was created as a knowledge representation framework for intelligent systems. The main idea behind TMK is that effective reasoning about a system is supported by a description of both *what* a system does and *how* it works. TMK models are referred to as *functional* models because they focus on describing and decomposing the functions of systems. Because TMK models were designed for use in artificial intelligent systems, existing versions of the TMK language require the enormous precision and formality necessary to support automatic processing. However, the basic idea behind TMK, that functional models of systems are useful for reasoning about those systems, also seems appropriate for information to be used by humans. This paper presents a new version of TMK which is less formal than the existing versions of the language. This semi-formal version of TMK is intended to be used as a software architecture modeling language for use by human software engineers.

Human software engineers do a great deal of reasoning about software systems, but their knowledge of these systems is often implicit and may not be easily verbalized. Consequently, a wide variety of notations have been proposed to enable the encoding of this knowledge [3, 1]. However, even with extensive experimentation in this field, there is still only a limited degree of understanding as to what constitutes a useful description of a software system.

Task-Method-Knowledge (TMK) models were originally created as a formalism for allowing intelligent systems to reason about software [2, 4, 5, 6]. These intelligent systems use the information encoded in TMK to manipulate and alter software modeled within the TMK framework. The fact that these systems are able to do so, strongly implies that the content of TMK models is useful for reasoning about software. This suggests that the content of a TMK model may be useful to a human software engineer who is attempting to manipulate software. This paper presents a preliminary attempt to provide a representational framework whose content is drawn from existing intelligent systems research but whose form is specifically tailored for production and comprehension by humans.

A TMK model encodes three major kinds of information:

- *Tasks* are what a system or a piece of a system does, i.e., a functional specification of a piece of computation.
- *Methods* are how a system or a piece of a system works, i.e., a behavioral description of a piece of computation.
- *Knowledge* is the information that a system or a piece of a system processes.

The definitions of tasks, methods, and knowledge are tightly interdependent. A description of what a system does necessarily involves referring to the information that it processes (since a piece of software, by definition, is an information processing mechanism). Furthermore, in TMK describing of how a system works involves describing what the pieces of that system do. Thus tasks and methods are defined in a hierarchy in which tasks relate to the methods which accomplish them and methods relate to the combination of tasks which together accomplish them. Consider, for example, the task of searching for an item in a list of items. One method for accomplishing this task involves stepping through the list one entry at a time, checking each entry to see if it is the one being looked for (i.e., a sequential search). The actions in this method, "stepping through the list" and "checking each entry" are themselves tasks. Those tasks, themselves, may have their own methods which accomplish them. At the bottom of the hierarchy of tasks and methods are primitive tasks. These tasks are sufficiently simple that it is not productive to break them down further into simpler tasks. For example, if a search is being done over a list of integers, checking an entry in that list simply involves checking to see if those two integers are equal; this action would be represented as a primitive task in TMK.

An example of a system modeled in the semi-formal TMK language is provided at the end of this document. Figure 1 provides a diagram of the tasks and methods in this model. The system modeled in this example is a simple hypothetical meeting scheduling program.

The top level task of the agent is the task of scheduling a meeting. It has one method which it uses, that of enumerating a set of slots and checking those slots against the schedules. The slot enumeration mechanism sets up two subtasks: finding a slot to try and checking that slot against the list of schedules. It also describes the order in which those subtasks are executed: it finds a slot and then checks that slot repeatedly until either it is unable to find a slot (i.e., the meeting scheduler has tried every slot it can and thus has failed) or a slot has been successfully checked against the participant's schedules (i.e., the meeting scheduler has found an appropriate slot and thus has succeeded). There are two different methods for finding a slot in this simple meeting scheduler. The first simply finds a the first time slot which hasn't yet been tried. The second randomly selects a time slot which hasn't been tried. This meeting scheduler is set up to always use the first method whenever the length of the desired meeting is less than 60 minutes and the second method whenever the length is more than 60 minutes.[1] The task of checking a slot against a set of schedules has one method, which simply checks the slot against each schedule, one at a time.

A description of a task in TMK involves five different pieces of information:

Input: The set of knowledge items taken as input to this task. Each item of input is specified by a label and the name of the abstract concept which that item instantiates. For example, the input for the schedule-meeting task includes an item (meeting-length Length) which encodes the length of the meeting to be scheduled. An input may also include a list (or another data structure such as a stack or queue) containing elements of a specific concept. For example, the inputs to the find-slot task include an item (slots-already-tried (List OF Time-Slot)) indicating that this input is a list of time slots.

Output: The set of knowledge items produced as output by this task. The format for output entries is similar to the format for input entries. An example of an output is the item (meeting-time-slot Time-Slot) for the schedule-meeting task which indicates that this task produces a time slot in which a meeting is to be held.

Given: A logical assertion which must hold in order to execute the task. This assertion can refer to the input knowledge items, the relationships specified in the knowledge description (see below), and standard logical / boolean operations such as and, or, not, =, etc. The assertion is presented using standard prefix notation.

Makes: A logical assertion which must hold after the task is successfully executed. This assertion can refer to both input *and* output knowledge as well as the known relationships and standard operations. An example of such an assertion from the schedule-meeting

[1] As noted in the description of the sequential method, the motivation for this distinction is that repeated use of the scheduler using the sequential slot choosing mechanism tends to cluster meetings together while using the random slot choosing mechanism tends to spread meetings out more; this meeting scheduler design assumes that short meetings should be clustered together (so that work isn't constantly being interrupted by many short meetings) but that long meetings should be more spread apart (because attending several consecutive long meetings can be exhausting).

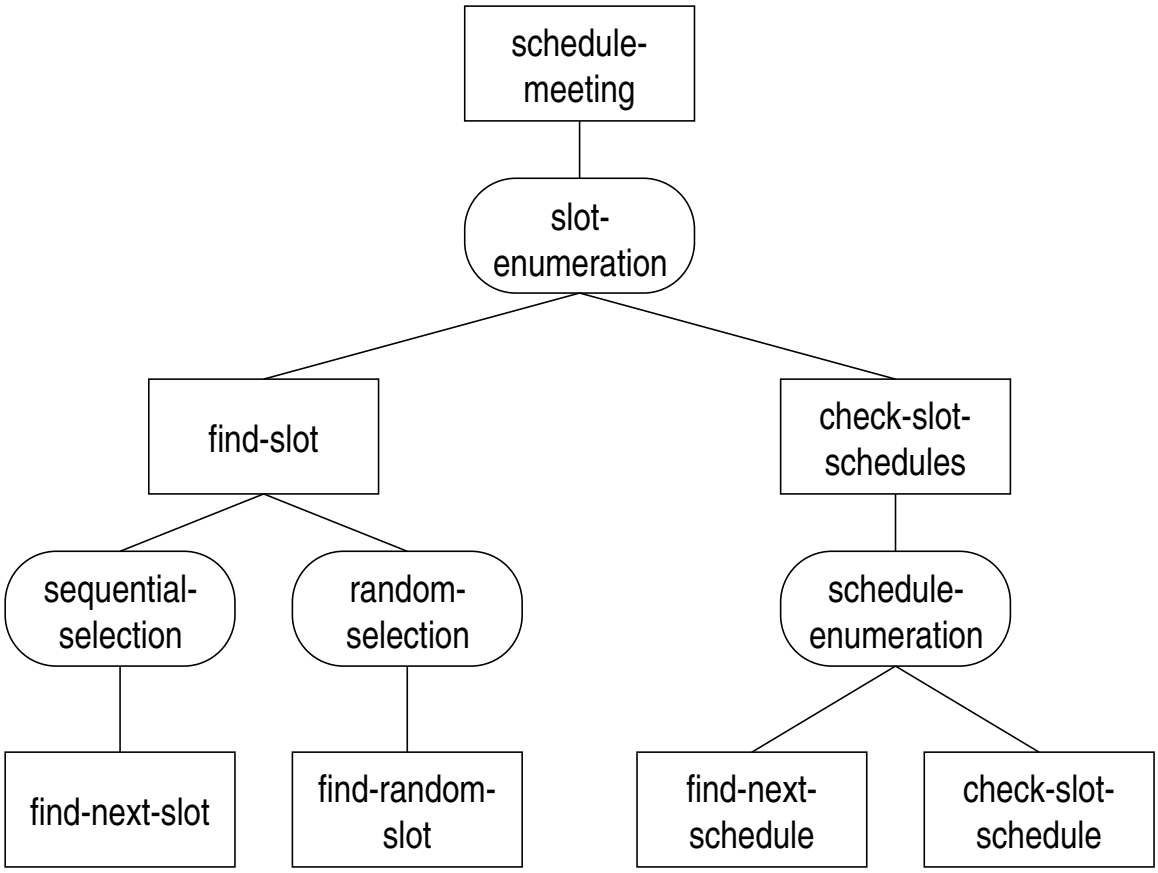

Figure 1: The tasks and methods of the example meeting scheduling system. Rectangular boxes indicate tasks while rounded boxes indicate methods. Lines from a task down to methods indicate that the task is accomplished by selecting one of the methods. Lines from a method down to tasks indicate that the method is accomplished by combining the tasks.

is (and (= (length meeting-time-slot) meeting-length) (fits meeting-time-slot participant-schedules)). This assertion states that if the schedule meeting task successfully completes, it will produce a time slot whose length is equal to the given length and which fits into the given schedules. The length and fits relationships are described at the end of this TMK model.

By-Method: The set of methods which accomplish the task.

Each of the above entries is optional if the information which they encode is not relevant. For example, the check-slot-schedules task does not produce any output (it either succeeds or fails and this success or failure determines whether the current meeting-time-slot is used or discarded); consequently, this task has no output entry. Also, many tasks are missing given and/or makes slots because there are no easily specified logical assertions which are required to hold before or after their execution.

Some tasks are primitive tasks, i.e., tasks which are not further decomposed into methods. Primitive tasks do not have a By-Method entry in their descriptions. Instead they have a brief text description of what it is that they do.[2] For example, the find-random-slot primitive task selects random slot which has not already been tested and fails if all possible slots have already been tried.

A task in TMK which isn't primitive must have at least one method specified for it. A description of a method in TMK encodes three different pieces of information:

Provided: A logical assertion which must hold in order to execute the method.

Additional-Results A logical assertion which must hold in order after the method is successfully executed.

Subtasks: The tasks which are be accomplished in order to perform the method and the order in which they are performed. The ordering of subtasks of a method can be specified using pseudo-code, as in the example methods.

The provided and additional-results entries in a method are similar in form to the given and makes slot in a TMK model but serve a different purpose. The purpose of the given and makes entries of a task is to describe what the task is intended to accomplish, i.e., what the point of the task is. In contrast, the provided and additional-results generally describe incidental constraints that are a by product of the particular way in which this task is accomplished. Consider a part of a system which downloads a file from a machine on the network. The point of this piece of the system is that *given* that the file is on one machine, it *makes* the file be on the other machine. One method for performing this task might be an FTP transfer. This method would operate *provided* that the machine with the file is running an FTP server, and might have an *additional result* that an entry is left in the server's log file. Leaving an

[2] In the more formal, AI versions of TMK, descriptions of primitive tasks refer to precise specifications of what these primitives do, typically in the form of source code in some general purpose programming language such as LISP.

entry in the log file is not the point of doing a file transfer, it's just something that happens as a result of that particular method of transferring a file.

Concepts and relationships describe the kinds of entities and connections which exist in the domain in which your system operates. For example, the meeting scheduling domain involves concepts such as time slots and schedules and relationships such as the fact that a time slot has a particular length or that a particular time slot fits into a particular schedule. Recall that concepts are used to fill the input and output entries of a task while relationships are used for the given, makes, provided, and additional result entries in tasks and methods; consequently it is important to specify all of the concepts and relationships needed for those entries. In the semi-formal TMK notation, concepts and relationships are defined simply by text descriptions, as in the example.

Acknowledgments

This work has benefited from discussions with the MORALE group, especially Ashok Goel and Spencer Rugaber. Effort sponsored by the Defense Advanced Research Projects Agency, and the United States Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-2-0229. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, or the U.S. Government.

Appendix: A Simple Meeting Scheduling System

```
Task: schedule-meeting
 Input: (meeting-length Length) (participant-schedules (List OF Schedule))
        (meeting-constraints Time-Constraints)
 Output: (meeting-time-slot Time-Slot)
 Makes: (and (= (length meeting-time-slot) meeting-length)
             (fits meeting-time-slot participant-schedules))
 By-Method: slot-enumeration

Method: slot-enumeration
 Subtasks: REPEAT
             IF find-slot THEN
               IF check-slot-schedules THEN
                  SUCCEED
               ELSE
                  NEXT
             ELSE
```

```
               FAIL

Task find-slot
 Input: (meeting-length Length) (meeting-constraints Time-Constraints)
        (slots-already-tried (List OF Time-Slot))
 Output: (meeting-time-slot Time-Slot)
 By-Method: random-selection sequential-selection

Method: sequential-selection
 Provided: (< (length meeting-time-slot) 60-minutes)
 Subtasks: find-next-slot
 Comment: [The sequential slot choosing mechanism tends to cluster
           meetings together; this meeting scheduler design assumes
           that short meetings should be clustered together (so that
           work isn't constantly being interrupted by many short
           meetings) but that long meetings should be more spread
           apart (because attending several consecutive long meetings
           can be exhausting).  This is why the sequential-selection
           method is only applicable to meetings which are 60 minutes
           or shorter.]

Primitive Task: find-next-slot
 Input: (meeting-length Length) (constraints Time-Constraints)
        (slots-already-tried (List OF Time-Slot))
 Output: (meeting-time-slot Time-Slot)
         (slots-already-tried (List OF Time-Slot))
 [This task chooses the soonest possible time slot which satisfies the
  constraints and is not in the slots-already-tried list.  It fails if
  there is no such slot.  This task also adds the chosen slot to the
  slots-already-tried list.]

Primitive Task: find-random-slot
 Input: (meeting-length Length) (constraints Business-Hours)
        (slots-already-tried (List OF Time-Slot))
 Output: (meeting-time-slot Time-Slot)
         (slots-already-tried (List OF Time-Slot))
 [This task chooses a random slot which satisfies the specified
  constraints and is not in the slots-already-tried list.  It fails if
  there is no such slot.  This task also adds the chosen slot to the
  slots-already-tried list.]

Task: check-slot-schedules
```

```
 Input: (meeting-time-slot Time-Slot)
        (participant-schedules (List OF Schedule))
 By-Method: schedule-enumeration

Method: schedule-enumeration
  Subtasks: REPEAT
              IF find-next-schedule THEN
                IF check-slot-schedule THEN
                   NEXT
                ELSE
                   FAIL
              ELSE
                SUCCEED

Primitive Task: find-next-schedule
 Input: (participant-schedules (List OF Schedule))
        (schedules-already-checked (List OF Schedule))
 Output: (participant-schedule Schedule)
         (schedules-already-checked (List OF Schedule))
 [This task chooses the first schedule in the list which hasn't
  already been checked.  It fails if all of the schedules have been
  checked.]

Primitive Task: check-slot-schedule
 Input: (meeting-time-slot Time-Slot) (participant-schedule Schedule)
 [This task checks to see if a slot fits into a given schedule.  It
  succeeds if the slot fits and fails otherwise.]

Concept: Length
 [This is a length of time.  It is represented as a number of minutes.]

Concpet: Time-Slot
 [This is an interval of time.  It is represented as a start time and
  an end time; these times are represented as a day plus an hour and a
  minute using 24 hour time (e.g., Tuesday, 14 hours, 23 minutes means
  Tuesday 14:23, i.e., Tuesday 2:23PM).]

Concept: Schedule
 [This is a specification of what times an individual is already
  busy.  It is represented as a set of slots indicating existing
  scheduled events.]
```

```
Concept: Time-Constraints
 [This is a general specification of times in which something can
  happen (e.g., when a meeting may be scheduled).  A typical value for
  a Time-Constrains object is Monday - Friday, 9:00 to 17:00 (i.e.,
  normal business hours.]

Relationship: length (Time-Slot)
 [This relationship computes a Length associated with a given time
  slot.  For example a time slot Tuesday 10:30 - Tuesday 11:00 has a
  length of 30 minutes.]

Relationship: fits (Time-Slot Schedule)
 [This relationship indicates that a particular time slot fits into a
  given schedule, i.e., that the slot does not conflict with any of
  the slots in the existing schedule.]
```

References

[1] Paul C. Clements. A survey of architecture description languages. In *Eighth International Workshop on Software Specification and Design*, Germany, March 1996.

[2] J. William Murdock and Ashok K. Goel. A functional modeling architecture for reflective agents. In *Proceedings of the AAAI98 Workshop on Functional Modeling and Teleological Reasoning*, Madison, Wisconsin, July 1998.

[3] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[4] E. Stroulia. *Failure-Driven Learing as Model-Based Self Redesign*. PhD dissertation, Georgia Institute of Technology, College of Computing, December 1994.

[5] Eleni Stroulia and Ashok K. Goel. Redesigning a problem-solver's operators to improve solution quality. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 562–567, San Francisco, August 23–29 1997. Morgan Kaufmann Publishers.

[6] Eleni Stroulia and Paul Sorenson. Functional modeling meets meta-CASE tools for software evolution. In *Proceedings of International Workshop Software Program Evolution*, 1998.