Modeling Computation: A Comparative Synthesis of TMK and ZD

J. William Murdock
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
`murdock@cc.gatech.edu`

April 16, 1998

Technical Report GIT-CC-98-13

Abstract

How do we build computational models of computation itself? This paper analyzes two such modeling languages from the perspective of building a third language which integrates their advantages. The ZD language [Liver & Allemang, 1985, Allemang, 1997] models software components and configurations in support of verification and reuse. The TMK language [Stroulia, 1994, Goel et al., 1996, Griffith, 1997] models intelligent systems in support of failure-driven learning and other reasoning tasks. I claim that these objectives are ultimately different perspectives on the same problem, thus suggesting that the integration of the languages is possible. I further claim these different perspectives do lead to substantive differences in capabilities, thus suggesting that the integration of the languages may provide significant benefits over either language viewed in isolation.

Introduction

Consider a computer-based intelligent system which contains a model of its own reasoning and processing. Since the system is implemented as a piece of software, the model that it contains can be viewed as a software architecture. The language in which this knowledge is represented is thus, in some sense, an architectural description language. *The issue of intelligent reflection can be reduced to the issue of software architecture.*

Now consider an arbitrary computer program along with a tool for representing and manipulating an architectural description of that program. The program and the tool together can be viewed as a larger system which is capable of some processing of a representation of some part of itself. Awareness and understanding of self is clearly one of the defining characteristics of intelligence. The program and the tool together are thus, in some sense, a reflective intelligent system. *The issue of software architecture can be reduced to the issue of intelligent reflection.*

Note that the fact that these two fields of study reduce to each other does not make their study redundant. Physics and philosophy reduce to each other since each seeks to describe the nature of the universe. However, they apply distinct perspectives and methodologies to this study and have thus developed distinct and often complementary insights. Thus Descartes' mechanical philosophy helped inspire Newton's theories of forces and optics which, in turn, helped inspire Kant's theories of reason. Similarly, one can hope for the studies of software architecture and intelligent reflection to inform each other. This hope motivates the comparative synthesis presented here.

This report examines ZD, a language for describing software architectures, and TMK, a language for modeling reasoning processes. It does not provide a full description of either of these languages. For descriptions of this sort, see earlier work such as [Allemang, 1997] and [Stroulia, 1994]. Instead, this report examines details of the languages as they provide interesting and useful comparisons and contrasts with each other.

Examples

The ZD tutorial [Allemang, 1997], *q.v.*, presents as an example a time card processing system. This system is divided into three major pieces: an employee system on which time cards are recorded, a time account management system on which the time cards are processed, and a network which transfers data between the two. This system provides the primary motivating example for this paper. Three descriptions of this system are provided as appendices at the end of this document:

- a transcription of the example in ZD taken directly from the text of [Allemang, 1997]
- a rough translation of the example into TMK as it appears in the actual code for REFLECS and Autognostic-on-Router
- a prototype translation into a proposed rough draft of a new language, called TMKL, which combines elements of both of the above

The prototype here is intended to be more of a starting point for basing a discussion on then any sort of serious proposal at this point; I've really just started working on this. There are large portions of TMKL which need serious refinement and more large portions which I haven't even seriously considered yet.

Note that neither the TMK nor the TMKL example have been debugged except by inspection. They are both likely to contain assorted errors and are provided as rough illustrations only.

Terminology

Below when I refer to TMK and ZD, I refer to the versions described in [Stroulia, 1994], in which the name SBF is used in place of TMK, and [Allemang, 1997] respectively, except where explicitly stated otherwise. The new language which we are creating has been tentatively named TMKL for "TMK Language" (thus emphasizing the interest on language design issues *per se* in this project). When I refer to TMKL below, I refer to the tentative version which I am currently proposing as of the last revision of this document.

As I understand it, the term "ZD" is used in the tutorial, both as the name of the language and as the name of the program which uses the language. Where this distinction is relevant below, I refer to the latter as "the ZD system."

It may be important to note that the ZD example which I used used the NQTHM prover throughout the model as the state description language. Consequently, some of the things that I say about ZD may really be artifacts of NQTHM. Points number #2.4, #3.3, and #5.4 are ones which I think this remark may be particularly relevant to.

Versions of TMK

Some additional clarification may be needed here with regards to the many different versions of TMK which have been developed. Some of these versions are enormously different. Below is a list of a few of the most important ones:

- The language of Autognostic, which is described in [Stroulia, 1994, Stroulia & Goel, 1994, Stroulia & Goel, 1995, Stroulia & Goel, 1996] and others, is considered to be the basic TMK language; again, this document will focus almost entirely on this version. Note that these papers generally refer to this language as "SBF" in analogy to the SBF language for modeling physical devices [Goel et al., 1997]. The term TMK, as used in this document, comes from related work and is applied to the same language which has been described as the SBF language.

- Interactive Kritik [Goel et al., 1996] uses a version of TMK as a language for describing what a system is doing. This version of TMK is inspired by the language of Autognostic but is intended to be read by humans (from a graphical display) rather than reasoned about by the computer system. As such, this version lacks the precision and formality of the other TMK versions.

- ToRQUE [Griffith, 1997] uses a version of TMK which was developed for this project. It generally resembles the version in Autognostic but is considerably simpler since the language is used simply to control processing and not to enable self-redesign.

- SIRRINE-1 is essentially an extension of Autognostic; the language used in this project is essentially the language of Autognostic.

- SIRRINE-2 [Goel & Murdock, 1998] is a relatively simple system which is being developed concurrently with the writing of this report. It is intended to be act both as a vessel for exploring very short term research goals and as a prototype for a new system to be based on the TMKL language being developed here. SIRRINE-2 uses a version of TMK which combines features of SIRRINE-1 and Autognostic with some of the capabilities described here as well as a number of relatively ad hoc shortcuts intended to allow very rapid development.

#1 Syntax

#1.1 TMKL syntax

One crucial issue involved in the definition of a language is the syntax used for specifying constructs. Clearly ZD is much cleaner in this regard than TMK is; the ZD code is almost all in the form:

```
(deffoo fooname
  :slot1 value1
  :slot2 value2
  ...)
```

In contrast, the TMK code is extremely heterogeneous syntactically and makes direct use of a variety of LISP constructs (*e.g.*, the domain concepts are implemented as "defclass" statements and the set of domain relations are defined using a setf statement). One possibility which I have strongly considered is the use of an entirely new syntax. The primary motivation for this approach involves making clear the distinction between the language and the implementation of the system.

On further consideration, however, a number of drawbacks to this approach present themselves. One is that that developing a parser for such a syntax could be time consuming, whereas with both the ZD and TMK approaches, the input can be specified entirely as macros and the parsing thus done by LISP. However, to some extent this is a disadvantage as well because it allows (or at least implies that it allows) users to "circumvent" the syntactic constructs that we want to provide and insert arbitrary LISP code; while this provides some power, it also seriously compromises TMKL's coherence as a language. Furthermore, while developing a parser may take some time, in the long run it would still be an extremely small fraction of the total project effort, particularly if the syntax was simple and precisely defined. It is this reasoning which originally lead us to develop a new syntax.

However, there are other drawbacks to the approach of developing a new syntax. The most serious, to me, seems to be that a new syntax would be unfamiliar to anyone we were presenting our work to. Users would have to learn the syntax; this is not a major limitation since users would certainly have to learn a great deal about the system before using it anyway. However, people reading papers or viewing presentations that use examples from TMKL would have to learn enough about the new syntax to understand the examples. This would often be very easy to do, but could potentially be a perpetual nuisance.

In contrast, the LISP-like syntax used in ZD is extremely wide spread within the AI community. For most contexts in which this work would be presented, one can simply show examples using this syntax and assume that the syntax is familiar / understood. Consequently, I have decided that this will be the basic approach for TMKL (at least, in this version). In the current version I have added two features which I feel are particularly relevant to this language: nested definitions and multiple definitions. I feel that these features are crucial here because the complexity of the models involved becomes unmanageable if every data item has to be defined in an individual statement.

The syntax here is still a rough draft; the meaning should be fairly obvious but the exact details aren't too important at this stage of development. Here's a quick BNF-like sketch of the syntax I'm using in TMKL; the meanings of these constructs should be pretty clear from the example.

```
<Model> := <Decl>*
<Decl> := "(" <def-Type> <PretypedDecl> ")"
        | "(" <def-Type> "(" ( "(" <PretypedDecl> ")" )+ ")"
<PretypedDecl> := [ <Name> ] ( <SlotName> <SlotValue> )*
```

```
<SlotValue> := <LiteralValue>
             | <Decl>
<LiteralValue> := <Constant>
             | "(" <LiteralValue> ")"

<def-Type> := "deftask" | "defmethod" | "defdomain" | "defconcept" | ...
<SlotName> is a keyword symbol
<Name> is a non-keyword symbol
<Constant> is a symbol, number or some other literal constant
```

Note that this syntax suffers from some ambiguity problems (from a formal language perspective); I don't think these will be too difficult to resolve, however.

#1.2 Off-the-shelf syntax

Another approach to syntax would be to use an existing general-purpose syntax such as ACME, KIF, KQML, CycL, UML, etc. While there are clear benefits to this approach (existence of off the shelf parsers, familiarity of the syntax to some potential users, easier interoperability with some existing systems, ability to document by reference, etc.), my personal feeling is that all of the aforementioned possibilities simply have too many features that we really don't want to have to deal with. If we could find an existing KR syntax which is very simple and easy to work with and still gives us all of the capabilities we need, I'd want to use it. ACME seems reasonably appropriate to many of our goals. I did some work converting from TMK to ACME in building the SIRRINE-1 system. ACME is somewhat specialized to represent the sort of information that we want to represent (*i.e.*, information about software architectures). We may want to give serious consideration to using ACME as a representation syntax. On the other hand, ACME seems to be very poorly suited to representing the information about knowledge / domains that we want to represent so we may could face serious challenges on that front. Furthermore ACME is largely intended to be an interchange format for sharing information between systems rather than a knowledge representation format for use within a system. ACME may be have more value to work using TMKL when viewed as a tool for interoperation rather than as a primary syntax for the language.

For now we have decided not to use an off-the-shelf syntax. All of these languages come with some "baggage" regarding how one is forced to think about representation. Using a specialized syntax seems more desirable for now.

#1.3 State description language

A related issue to that of the general syntactic mechanism is the specific details of the state description language; *i.e.*, how does one specify logical assertions in the language. It seems possible that we will want to use an off-the-shelf system for manipulating logical assertions (like NQTHM in ZD). For now, I have used essentially NQTHM-like (or for that matter, generally LISP-like) predicates in my example, without having explicitly committed to any particular SDL.

#2 Knowledge

#2.1 Ontologies

ZD has two closely related concepts: clusters and domains. For purposes of this discussion, I will refer to them collectively as "ontologies"; while this is a heavily loaded term, it roughly describes the issue at hand.

In ZD, clusters are specifically tied to one particular level in the service / implementation hierarchy while domains are independent of a particular system. As implemented in Autognostic, the only ontology TMK supports a single "domain"; declarations of domain concepts, etc. are made and are implicitly considered to be part of "the" domain. An equivalent ZD system would have one big global domain and each cluster would simply have a :use slot for that domain and no additional declarations.

The ZD approach of providing multiple interconnected domains seems to provide some substantial advantages. In particular, one of the real limitations of the TMK approach of having one general purpose "domain" is that tasks and methods wind up being much less "generic" than the structures in ZD because there inputs and outputs have to refer to the specific problem being solved. For example, consider the "transfer" requirement in the ZD example. It has variables like "source" and "destination" and then an explicit "defrelation" statement to link these to the problem specific variables "M-E" and "M-TAM". In contrast, in the TMK code, the "transfer" task refers directly to "M-E" and "M-TAM". This is because there is only one name space allowed.

Unfortunately, the ZD approach seems a little demanding; should it really be necessary for there to be a separate ontology for *every* level of *every* system? This isn't an enormous problem since ZD allows clusters to be defined as essentially extensions of another domain or cluster (with the :use slot).

In TMKL language, I have used a compromise between these two approaches. The language supports one type of ontology which is referred to as a "domain." Each task and method in the system then has a explicit pointer to the domain in which it is defined. Thus if we want to have one global ontology for the system, as in TMK, we simply define one domain and have all of the tasks and methods point to it. Similarly, if we want to have one ontology for each level, as in ZD, we simply define one domain for each level and have the appropriate tasks and methods point to the appropriate domains.

ZD allows ontologies to inherit values from higher level ontologies by assigning them a ":use" slot. This feature makes it possible to define nested hierarchies of ontologies. I have included a :use slot in the TMKL domains.

One of the problems that the support of multiple domains creates is the problem of dealing with inputs and outputs to a task which are only defined at lower levels than the task itself. Consider, for instance, the top levels of the IOP example from the tutorial. The top level service, submit, refers to the time systems, the time card, and the "loaded" relation, but does not refer to concepts such as machines, formats, etc. which are defined at lower levels. This raises an issue (which is not relevant to ZD but is to TMK) of how to define the inputs and outputs of such a task so that when the model is actually executed, the program will be able to assign values to variables. In TMKL, only the inputs and outputs which are present at a task's level of abstraction are referred to in that task's input and output slots. For example, the submit task refers to the TS-TAM variable as an input but not the M-TAM variable, even though both of these need to be defined in order to execute this task. This approach seems generally consistent with the way Autognostic handled these sorts of issues.

A more serious problem is the reverse situation in which lower level elements have to deal with information which is only defined at a higher level. Consider, for example, the export-comma service from the tutorial. This service does not refer to the machine which the system is being exported from; the requirement of the export service, that the card be PRESENT on the employee machine is ignored at that level, presumably because the analysis being done at that level does not depend on this. In TMK and TMKL, however, we need to connect this task to actual code which will actually accomplish the task; this code will need to have information such as the machine available to it. This is really a usage issue rather than a language issue. In TMKL, we represent the machine (and the PRESENT relation) in the TimeCard-domain, in contrast with the ZD code which makes no mention of these in the TimeCard service cluster.

#2.2 Domain integration

The existence of multiple domains within the language obviously demands some mechanism for integrating these domains. The ZD language appears to have two such mechanisms: relations and glue. The distinction, as I understand it, is that relations link services to the requirements that support them while glue links requirement clusters to the service clusters that implement them. Since TMKL allows domains to run across levels, these mechanisms which are tied to the ZD organization of information spaces will not work. Consequently, I have introduced into TMKL, a general purpose mechanism for interconnecting domains which I have labeled "correspondence."

One concern I have about domains is that while they add expressive power they can also be tremendously inconvenient. One of the real advantage of the TMK approach of one global domain for everything is that one never has to specify how the values in different sets of tasks and methods interconnect (since they all use the same name space). One could define correspondences for the domains globally (*e.g.*, say something like "all instances of 'data' in the TimeCard domain map to the term 'ETC' in the submission domain"). This would give us the ease of use of TMK but would completely eliminate the expressive power of having multiple domains (in effect there would be one global domain with several different name spaces to refer to concepts in the domain). In contrast, we could require that each task which interacts with another task of a different domain have its own separate correspondence between domains; in practice, however, this would lead to enormous duplication of code (*e.g.*, dozens of instances of the statement "the instance of 'data' in this task maps to the term 'ETC' in that task"). A compromise would be to allow multiple, potentially conflicting correspondences to be defined globally and require each task or method which interacts with another task or method in a different domain explicitly refer to the specific correspondence which is relevant. This is a far more concise approach but is still considerably bulkier than TMK. In the proposal, I have settled for an even more concise compromise: I allow multiple, potentially conflicting correspondences, but I require tasks and methods to explicitly state which correspondence they are using *only if* there is some conflict. In other words, if task A in domain P invokes method B in domain Q, A and B must specify which correspondence of P and Q they are using *only if* there is more than one correspondence of P and Q. If there is exactly one such correspondence, it is assumed that that correspondence is being used. This issue is discussed a little further in #5.6.

Another, more minor simplification which I have included is a rule that a correspondence of two domains which use the same label for some value automatically assumes that these labels refer to the same concept unless explicitly stated otherwise. I claim that this issue has no major theoretical implications (since the content of the correspondence is the same whether a link is present implicitly or explicitly) but in practice this is likely to be more concise (since programmers will usually only use the same terms if they really do have the same meaning). As an example, consider domains P and Q for which P has labels (a, b, c, d, e, x, y) which map in domain Q to the labels (a, b, c, d, e, m, n) in those orders but both domains also contain labels "z" which do not correspond to each other. We could define the correspondence of P and Q:

```
(defcorrespondence P-Q
 :domains (P Q)
 :unmap (z)
 :map ((x m) (y n)))
```

Thus the variables, a, b, c, d, and e, are assumed to map to each other in both domains but the variable z is explicitly stated not to map across the domains using the "unmap" construct. The expectation is that "unmap" would be used only very rarely (because the same label is rarely used for the same term) but is needed for expressive completeness.

Another thing to note is that the semantics of correspondences are complicated somewhat by the fact that domains (as stated in #2.1) support inheritance hierarchies so that domains may be unified at multiple levels. I have assumed here that domains which inherit values which are unified at higher levels also inherit this correspondence. As an example, consider domains P1, P2, Q1, and Q2 in which P2 isa P1, and Q2 isa Q1. A variable "r" is defined in P1 and a variable "s" is defined in Q1. There is also a correspondence mapping r to s:

```
(defcorrespondence P1-Q1
 :domains (P1 Q1)
 :map ((r s)))
```

In this example we have a task in domain P2 which inherits the variable r and outputs a value in that variable. Immediately following that task we have another task in domain Q2 which inherits the variable s and inputs a value in that variable. The variables r and s in these tasks do refer to the same values even though there is no explicit correspondence of P2 and Q2 because the variables inherit their correspondence.

I have specified three features which greatly simplify the specification of correspondences: assumed use of unique correspondences, assumed equivalences of identical labels within a correspondence, and inheritance of correspondence. The three of these (especially taken together) could potentially require a complex and potentially costly procedure to generally identify which labels are equivalent to which other labels. However, it seems obvious to me (I think) that these issues can all be resolved at the time that the model is loaded into memory. Therefore, this issue would not have to complicate or slow down the actual use of / reasoning about the model.

#2.3 Items and relations

In the mechanism for defining data items, TMK makes a strong distinction between data items (*i.e.*, domain concepts / information types) and relationships between these items (*i.e.*, domain relations / semantic relations). In contrast, ZD defines each of these things using ":declare" and ":types" lines within a given cluster or domain, with the only distinction is that some terms have a functional data type while others have a declarative type. I am of two minds on this issue. The ZD stance seems more parsimonious; on the other hand, there does seem to be a rather significant difference between the two.

I think the crucial question here is do we want to be able to specify relations as actual inputs to a task / service. For example, we could envision a "filter" task which takes as input a unary relation and a set of elements and outputs the units in the set for which that holds. In TMKL, I assume that we might want this capability and thus use the ZD approach of making relations be defined as a type of variable.

#2.4 Variability

Consider the issue of constant values and variable values. TMK seems to typically involve data items being variable and relationships being constant.

In the tutorial, it seems that ZD requires data items to be constant and relationships to be variable. For example, consider the following slots of the submit service from the ZD tutorial:

```
:If (LOADED TS-E ETC)
:ToMake (LOADED TS-TAM ETC)
```

The :ToMake slot indicates that at the end of the execution of this service, the statement (LOADED TS-TAM ETC) should be true, even if it wasn't before. Clearly something must change during submit: either the value of TS-TAM and/or ETC must change *or* the truth table for the LOADED predicate must change. In this case, the obvious interpretation is that the LOADED predicate changes; we are loading some data onto a system, not redesignating the system to one which already has the data or redesignating the data to some data which is already on the system. All of the examples in ZD seem to have this property, *i.e.*, that the terms remained fixed while the predicates change.

However, other work on ZD seems to suggest the opposite conclusion. [Liver & Allemang, 1985] presents an example of a stack in which a transition occurs from `S.top = X` to `S.top = X+1`. Clearly the intent here is that `S.top` has changed, not the definition of equality. It may be that this issue simply does not need to be fully resolved for the kinds of tasks that ZD is intended to address; *i.e.*, that the kinds of analysis done in ZD simply need to track which assertions are true at the different states but do not need to know *what* changed to make these things true.

Now consider, a fragment of the top level task in REFLECS, get-to-box:

```
:given (current-position goal-point)
:makes (current-position)
:semantics ((close-to-goal current-position nil goal-point nil))
```

The :semantics slot here indicates that at the end of the execution, the statement (close-to-goal current-position goal-point) should be true, and the :output slot indicates that this should be done by changing the value of the current-position variable, not, obviously, by changing our definition of closeness of points.

Unlike with ZD, it is clear that TMK and TMKL do need to make clear what changes during a task. One of the reasons for this is that these languages require the ability to *run* the model and thus need to actually allocate variables for things that change, etc. Another is that these languages support redesign of the system. Consequently there must be some indication of what elements do not change across or within executions because those elements are potential candidates for system redesign. For example, the fact that Autognostic does not allow variable predicates means that any predicate in the system may be altered by the redesign mechanisms.

In TMKL, I allow both concepts and predicates to be variables. The issue of whether something is constant is specified implicitly by whether it is the output of any task. See #5.7 for a further discussion of this issue.

#2.5 Concepts and types

Data items in ZD (*i.e.*, in NQTHM which provides the primitive for knowledge representation in the current ZD examples) are specified as one of a small set of primitive types or as restricted subclasses of these types (*e.g.*, the "form" type defined in the interchange-formats domain); the language doesn't directly support complex data structures, *i.e.*, there is no mechanism defined for specifying abstract data types. It is possible to implicitly define complex variables using numbers (or other basic types) as virtual nodes in a semantic network and functions as virtual edges. For example, one could define a person (*e.g.*, for a personnel record) which has an address, including street, city, and state; a phone number; and a salary as:

```
(defdomain personnel
  :types ((person :isa number)
          (address :isa number))
  :declare ((is-person :type (PRED 1))
            (person-has-address :type (FUN 2))
            (person-has-phone :type (FUN 2))
            (person-has-salary :type (FUN 2))
            (is-address :type (PRED 1))
            (addres-has-street :type (FUN 2))
            ...))
```

This system works, and seems to have sufficient power to handle most situations. However, it strikes me as extremely cumbersome for general use. Furthermore, it seems to lack the capability to divide knowledge items into single, compact representations. This would be particularly tricky to deal with in an environment in which we were explicitly specifying inputs and outputs to a task (as per #3.2); when we refer to a particular person as being the input to a task, we want to mean that the entire data item is the input, not just the particular number which corresponds to that person in the network.

In contrast, TMK has a somewhat confusing combination "domain concepts" and "information types". In theory, the distinction here seems to be that "domain concepts" are abstract, system independent data types while "information types" are abstract, system dependent data types. The input and output specifications of a task are supposed to be general, abstract definitions of what sorts of information they take and produce; thus the values specified in the "input" and "output" slots are the names of information types.

For example, in REFLECS, a robot control system in Autognostic, there is a domain concept "d-location" which specifies a physical location, defined as:

```
(defclass d-location (domain-concept)
    ()
  (:default-initargs
    :name 'd-location
    :identity-test 'spatially-close
    :attributes (list
                 (make-instance 'attribute
                                :name 'x-coordinate
                                :function '(lambda(x) (first x))
                                :type 'number)
                 (make-instance 'attribute
                                :name 'y-coordinate
                                :function '(lambda(x) (second x))
                                :type 'number))
   ))
```

As you can see, a domain concept roughly corresponds to a data type, similar to a value in a :types slot in a ZD domain. More precisely, a domain concept is a class such that an instance of that class would be a *description* of a type of value. In this case, a location would be a list of two numbers specifying a set of coordinates. An instance of d-location would be a description of such a value, specifying what the slots are and how they can be accessed. There are a number of different locations which REFLECS reasons about. For example, the input to the main task of REFLECS, takes in the location of the robot, "current-position", and the location of the goal, "goal-point", and outputs the location of the robot, "current-position", with the semantics that the "current-position" on output is close to the "goal-point". This leads to the information type declarations:

```
(defclass current-position (d-location problem-variable)
    ()
  (:default-initargs
    :name 'current-position
    :input-to (find-where-info-is-input known-tasks 'current-position)
    :produced-by (find-where-info-is-output known-tasks 'current-position)))

(defclass goal-point (d-location problem-variable)
    ()
  (:default-initargs
    :name 'goal-point
    :input-to (find-where-info-is-input known-tasks 'goal-point)
    :produced-by (find-where-info-is-output known-tasks 'goal-point)))
```

[As a side note here: the code here uses the name "problem-variable" but the dissertation refers to these as "information types". Presumably "problem-variable" is an older term.]

In this example, at least, an information type appears to essentially be a variable declaration, similar in purpose to a value in a :declare slot in a ZD domain or cluster (except as noted in #2.4). More precisely, an information type is a subclass of a domain concept such that an instance of that class would be a *description* of a variable which would hold a value such that an instance of the domain concept would describe the value. This strikes me as confusing.

Table 1 shows some equivalences between the terms (for Autognostic and TMK, there are two different names listed for each slot, these correspond to the item / relationship distinction described in the previous bullet point; the ZD and TMKL terms associated with those slots capture the content of both varieties of information).

Table 1: Equivalences between knowledge constructs

ZD	TMK	Autognostic Code	TMKL
:types	domain concept domain relation	domain-concept domain-relation	:concepts
:declare	information type semantic relation	problem-variable relation	:variables

The differences in terminology are not really the major issue here, however. In TMKL I have :concepts and :variables as having largely the same *external meaning* as ZD's :types and :declare slots, *i.e.*, as declarations of data types and variables. The precise meanings of the TMK terms (as classes who's instances are descriptions of instantiations) is confusing. In contrast, I have their *internal content* as being largely based on TMK; they have slots for things like attributes, identity tests, etc. I claim that the kind of reasoning we do requires this sort of information.

#2.6 Domain constraints

ZD and TMK both have notions of domain constraints; facts that are always true within a domain. TMK domain constraints are extremely limited; they essentially can only be statements of the form if Y(A,B) then Y(B,A) [whenever Z(A,B)]). My impression is that the TMK representation for domain constraints seems useful for a few specific kinds of issues, but do not really form a general purpose framework for handling constraints.

ZD domain constraints are arbitrary statements in the SDL and thus much more powerful. ZD supports constraints both over specific variables and over universally qualified values; in order to constrain a single type, one must have a tag which explicitly labels values as having that type (using functions as slots, as per #2.5. There are some quirks in how ZD handles constraints over user-defined types, but these are closely tied to the use of NQTHM as an SDL and would change if a different SDL were used. In TMKL, I essentially use the ZD approach to constraints; the specific details involved do depend, however, on the choice of SDL (see #1.3).

#3 Tasks and Methods

#3.1 Task structures

At first glance, tasks and methods in TMK seem to correspond roughly to services and implementations in ZD; tasks and services are both essentially something that the system does and methods and implementations are both essentially how the system does it.

However, there seems to be a pretty significant distinction between the two and it seems to revolve around the presence of requirements in ZD. I think the following point was discussed at our first meeting, but I think this example makes things clearer, at least for me:

Consider a simplified version of the classic monkey and bananas problem in which a monkey wants to get some bananas hanging on the ceiling and a box is already present under the bananas. TMK would divide this up:

```
Task:             acquire-banana
                        |
Method:         banana-acquisition
                  |      |       |
Subtasks: go-to-box get-on-box grasp-banana
```

In contrast, ZD adds an additional level of abstraction between the method and the lower level tasks which accomplish this method:

```
Service:          acquire-banana
                         |
Implementation: banana-acquisition
                  |        |      |
Requirements: be-at-box be-on-box have-banana
                  |        |       |
Services:     go-to-box get-on-box grasp-banana
```

The significance of this distinction only becomes clear when we examine capabilities which can be accomplished in more than one way. For example, imagine that the monkey can get to the box two different ways: by walking and by jumping. In TMK, we would say that the task go-to-box has two different methods:

```
                         |
Subtask:              go-to-box
                      |      |
Method:  jumping-to-box    walking-to-box
            |      |         |    |     |
Subtask:  jump1 jump2    step1 step2 step3
```

In contrast, as I understand the ZD approach, alternative capabilities are implemented as not as different implementations for a single service but as different services which can accomplish a specific requirement.

```
                         |
Requirements:         be-at-box
                      |       |
Services:    jump-to-box   walk-to-box
                |   |        |   |  |
```

Thus it is more reasonable comparison to say that a ZD service is like a TMK task with a method already selected and that a ZD implementation is like a TMK method with only abstract specifications of the subtasks.

So which approach is "better"? Well, it seems fairly obvious to me that it is generally possible to translate between the two, more or less. The ZD approach provides one additional level. As I understand it, the primary purpose of the additional layer is to allow sets of service / implementation / requirement data items to be viewed as modularized "components" in a reuse framework; the requirements provide abstract specifications of the lower level services which enables a user to easily remove a service which is fulfilling some requirement and replace it with another service which can also fulfill that requirement.

TMKL uses tasks and methods which mostly correspond to TMK tasks and methods rather than ZD services, implementations, and requirements. However, I'm still not certain that there is a convincing argument either way; this may be a fairly arbitrary decision.

#3.2 Inputs and outputs

In TMK, each task has an explicit specification of its inputs and outputs. In contrast, a ZD service may mention certain inputs and outputs indirectly (through the :if, :tomake, :provided, and :addres) slots, but does not require that a complete list be available. In fact, inputs and outputs for ZD services may not even be *defined* at the level of the service. Part of the issue seems to revolve around the fact that TMK models are actually executed. It is necessary to actually provide the input to the system in order to get the thing to run and furthermore, it is necessary to check to see if the input is available in order to monitor and respond to error conditions such as missing information. In TMKL, I do provide :input and :output slots for this reason; if we want to run the code, then we need to know what information to provide. However, information which is needed at lower levels but is not defined at the top level is only specified at those levels; the user will simply have to provide this information anyway or the system will report that the information is not available when it tried to execute these lower level tasks.

#3.3 Planning and problem solving

Here is a claim closely related to both #2.4 and #3.2 which seems to be extremely fundamental to the difference between ZD and TMK: ZD devices[1] are very much *planning* entities, while TMK tasks are very much *problem-solving* entities. From a functional perspective, services in ZD are defined very much like planning operators: given that some state holds, make some other state hold. In contrast, tasks in TMK are defined very much like problems: given some knowledge produce some other knowledge. It is clear that it is possible to translate between the two views: we can view a planning task as a problem-solving task where the output is a plan or we can view a problem-solving task as a planning task where the desired state is that some knowledge is in memory. Consider, for example, the top level task in Autognostic-on-Router, route-planning. Consider the fragment of the route-planning task definition:

```
:input (initial-point final-point high-zone)
:output (desired-path)
:semantics ((same-point desired-path initial-node initial-point nil)
            (same-point desired-path final-node final-point nil))
```

This is a very problem-solving oriented view: the route-planning problem says, given a start point, and end point, and a map, produce a path through the map which starts at the start point and ends at the end point. We can directly translate this into ZD (ignoring for the moment, the distinction between variable predicates and variable values discussed in section #2.4):

```
:if (null desired-path)
:tomake (and (same-point (initial-node desired-path) initial-point)
             (same-point (final-node desired-path) final-point))
```

This is a much more awkward definition. It is essentially saying that we start out not knowing the result and end up with a result which satisfies the stated conditions. In contrast, TMK has a serious problem modeling computational systems where the results involve doing something rather than producing some knowledge. Consider, for example, the primary task in REFLECS, get-to-box. A fragment of the definition:

```
:input (current-position goal-point)
:output (current-position)
:semantics ((close-to-goal current-position nil goal-point nil))
```

Intuitively this seems to me to be an extremely awkward definition. If this is read the way the Router example is, it says that given a position and a destination, determine a position near that destination. Really, this isn't what REFLECS does at all; what REFLECS does is *move* the robot from the start point to a point near the goal. ZD can express this much more naturally:

```
:if (at robot start-point)
:tomake (close robot goal-point)
```

[1] In particular, I am referring to the ZD devices in the tutorial here; those in [Liver & Allemang, 1985] seem to be more focused on computing values than on achieving states; in this sense they are more like the examples in Autognostic than those in the tutorial are.

This is read: given a robot at the starting point, move it to the ending point. Again here the issue seems to be somewhat conflated with the issue presented in section #2.4, *i.e.*, that in TMK we are changing the values but leaving the predicates alone while in ZD we are changing the predicates but leaving the values alone. In fact, the ZD approach of changing only the predicates is very similar to the way classical planning systems (*e.g.*, STRIPS) handles values.

From these examples as well as my own experience with Autognostic and SIRRINE, it seems like the ideal solution to the planning vs. problem solving issue depends heavily on the system being modeled. In particular, the following systems that we have worked on seem very amenable to the TMK problem-solving view: Router, Kritik, and the meeting scheduler. In contrast, the following problems seem much more amenable to the ZD planning view: the controller in REFLECS, the elevator scheduler, Mosaic, and the IOP example from the ZD tutorial. Each of these systems *can* be forced into the TMK framework (and this has been done with REFLECS and Mosaic) but in each case, it comes out as something of a "hack."

If it is true that some systems work better with one view and some work better with the other, perhaps we should come up with a system which combines the views. I think that this is partially be solved by allowing both relations and data items to be implemented as variables which can be both inputs and outputs of tasks as discussed in #2.3 and #2.4. Since relations can be outputs of the system, we can define tasks which make some state in the world true. Thus in TMKL, I represent the submit task (from the IOP example) using:

```
:input (TS-TAM TS-E ETC LOADED)
:output (LOADED)
:given (LOADED TS-E ETC)
:makes (LOADED TS-TAM ETC)
```

Note that this has some additional content over the representation used in ZD; it not only says that at the end, the (LOADED TS-TAM ETC) relation holds, it also says that it does this by changing what is and is not loaded, rather than changing TS-TAM or ETC.

#3.4 Preconditions and postconditions

Consider the syntactic mechanisms for specifying the relations that hold before and after some functional element has been executed. TMK has a slots labeled :under-condition and :semantics but the syntax used to specify them is relatively ad-hoc, clumsy, and lacking in expressive power. In contrast, ZD uses slots referred to as :if and :tomake which use prefix notation predicate calculus to specify relations (*e.g.*, "(and (in-form format ETC) (PRESENT M-E ETC))"). This seems far more useful.

It is interesting to note that in TMK, tasks have both under-condition and semantics while methods have under-condition but not semantics. My impression is that this is a fairly ad hoc trait resulting from a quirk in the usage of these slots. In contrast, ZD has both :if and :tomake slots at the levels of both services and requirements. In TMKL, I use the ZD syntax and make both :given and :makes slots present in both tasks and methods.

#3.5 Primitive computation

Here is a relatively minor issue: should primitive elements be defined at the level of tasks or at the level of methods? ZD does not seem to have a real commitment to this issue one way or the other since one can go down to whatever level one wants to in ZD and then stop. Since TMK is directly connected to the code, however, there has to be a bottom level which refers to an actual function in the program. In Autognostic, this level is tasks; tasks may either have methods which accomplish them or functions which do so. However, one could potentially use methods are bottom level elements. It seems to me like very little rides on this; I could make mildly convincing arguments either way. In practice, however, my experience suggests that the latter approach is moderately more convenient. Take, for example, the import task in the IOP system. The task of importing can be accomplished either by importing comma separated data or by importing tab separated data (both primitive operations in this model). In the TMK example, this is represented by methods (*i.e.*, importing-comma and importing-tab) which each have exactly one subtask (*i.e.*, import-comma and import-tab) which each refer to a procedure (*i.e.*, do-import-comma and do-import-tab). If we were to make methods be bottom level elements, the importing-comma and importing-tab methods would directly refer to the procedures which accomplish them (thereby leading to one less level in the model).

On the other hand, because tasks explicitly specify their inputs and outputs, it may be simpler to handle the parameters and return value(s) of the functions at the level of tasks. Autognostic handles this as follows: if a task lists inputs (A B C) and outputs (D E) and invokes function F, then F is a function of three arguments (A, B, and C, in that order) which returns a list of two arguments (D and E in that order). If we were to make methods be primitive, either we would need some explicit mechanism for mapping parameters and return value(s) to variables in the model or we would need to define inputs and outputs at the level of methods (which creates its own set of problems). For this reason, TMKL takes the TMK approach of making tasks act as the primitive elements of the model.

#3.6 Run time selection

The discussion in #3.1 ignores a distinct but relevant issue; that of whether these alternatives are selected at design time or at run time. ZD binds services to requirements at design time; the presence of multiple services which could potentially accomplish a given requirement indicates a set of *design* alternatives. TMK, however, binds methods to tasks at run time; the presence of multiple methods which could accomplish a given task indicates a set of *reasoning* alternatives. The TMK approach is tied to the fact that TMK is intended to be a language for describing multi-strategy reasoning systems. In order to retain this focus on multi-strategy reasoning, I have taken the TMK approach of binding alternatives at run time in TMKL.

#3.7 Terminology

Here is a minor terminological issue relating to #3.2 and other some other issues. TMK uses the terms “given” and “makes” to refer to the list of inputs and outputs respectively; this terminology is derived from the earlier SBF language for modeling physical devices. However, in this earlier language, the given and makes information did not specify what was coming into and going out of something but, rather, describes states of the world required by and resulting from that thing (and thus to the substances involved, only indirectly). This seems to correspond much more closely to “:if” and “:tomake” slots in ZD. While the content of these slots are present in TMKL, I have decided to use the names “:given” and “:makes” to continue the consistency with the SBF physical device language. Table 2 shows some equivalences between these terms.

Table 2: Equivalences between input / output slots

ZD	TMK	TMKL
	given makes	:input :output
:if :tomake	semantics under-condition	:given :makes

#4 Reasoning

#4.1 Phases

Autognostic essentially divides up its reasoning into three phases: monitoring, blame assignment, and adaptation. As I see it, ZD has little to say about adaptation (unless I'm missing something). However, I think there is a lot to be gained by examining the relationship between monitoring + blame assignment in Autognostic and the types of reasoning that is done in the ZD system.

#4.2 Monitoring

It looks to me like the state-transition lists in ZD can be viewed, to some extent, as predefined, user-encoded equivalents of the monitoring process in Autognostic. One major distinction is that the results of monitoring concretely describe the results of a single execution of the system while the ZD state-transition lists abstractly describe the results of any execution of the system. In the Time Card example, this distinction is relatively minor because there is only one path through the system.

#4.3 Blame assignment

Here is a bold proposition: "PPP in the ZD system corresponds to blame assignment in Autognostic. CCSP in the ZD system is simply assumed in Autognostic." I'm not really certain if this is a fair comparison but I would like to at least consider it. What I'm suggesting here is that the kind of dynamic analysis of the flow of values that is done in PPP is very similar to the analysis that Autognostic does in tracing through an execution to find a component that is broken. Conversely, the sort of static analysis that CCSP does in which components are checked to see if they could, in principle, be connected is not meaningful in Autognostic since the program has to start with a running (if flawed) program to begin with.

One of the major differences between blame assignment in Autognostic and PPP in ZD is the fact that because Autognostic's blame assignment process is driven by traces, the route that it takes in analyzing the system is dependent on the nature of a particular problem solving episode. In some cases, this can be a major disadvantage. In particular there seem to be three major limitations of doing blame assignment based on traces:

- In a system containing loops, a trace may be much larger than the model itself. The blame assignment process may take unnecessary time checking through many iterations of a loop to find a problem in the base case of a loop or even in an event that took place before the loop.
- It is possible that reasoning based on traces may divide analysis into pieces which are too fine. The PPP algorithm may be able to discover what are essentially "emergent" properties of the

system which cannot be determined from any one flaw but which arise from interactions of a variety of provisos. In many cases, it may also be possible to derive this sort of information from traces, but perhaps not all. Existing work with TMK has largely assumed a single point of failure, and thus not addressed this issue.

- Traces obviously conceal any flaws in those portions of a system which were not activated during the execution which generated them as well as flaws which aren't relevant to the particular data items used in that execution. This is generally not a problem in the task addressed by Autognostic, failure driven learning, since the reasoner is attempting to address a concern which is directly tied to a particular failure (and thus must have related to the trace which involved that failure). Other tasks, however, may require consideration of a larger range of features than are relevant to a particular trace.

Traces do, however, seem to provide some significant benefits. Two of the most significant benefits of traces are:

- For complex systems, traces may be much smaller than the models of the system itself. Even when they are larger, they may be much simpler since they follow a single path while the model as a whole typically involves many branching, intertwined paths. Thus reasoning may be much more efficient with relatively concise traces to provide localization.

- Reasoning using traces may detect flaws which are not explicitly accessible from the model.[2] A system may explicitly state a set of provisos which are internally consistent but which do not reflect the actual operation of a system. In these cases, a trace may be used to localize a flaw (and thus suggest a component for replacement) where the model itself does not indicate, even implicitly, any problem. Another way of stating this issue is that reasoning using traces may have weaker additional knowledge requirements; information about specific properties of lower level reasoning elements which would have been needed to detect a flaw using proviso propagation may be omitted because the problems that they cause at higher levels can be detected from the trace.

I would suspect that reasoning with TMKL will want to use elements of both Autognostic's blame assignment and ZD's PPP. While there are clearly benefits to the use of traces, there are also potential drawbacks.

#5 Additional Concerns

Everything discussed above is clearly still an "open issue", *i.e.*, nothing has yet been firmly resolved regarding the new language. However, I would like to briefly characterize a few specific issues which are particularly "open", *i.e.*, for which either I don't yet even have a preliminary proposal or I feel the existing proposal is clearly inadequate.

[2]Note that in general, any hierarchical functional model of a computational system will have to describe only *some* of the relevant properties of its inputs, outputs, and effects. If a model described *everything* about the system, it would typically have to be at least as long an complex as the system itself. This would eliminate most of the advantages of using a model in the first place; why use such a model when one can simply directly reason about the code itself? Thus the issue arises of how much information needs to be specified in any given model.

#5.1 Problem specification

Some mechanism is needed for specifying particular problems for the system in two different senses. Firstly, we need a mechanism for telling a system to execute the model for a specific set of inputs (*e.g.*, "given the map of GIT, find a route from 10th & Center to Ferst & Dalney"). Autognostic does not have an explicit encoding of problems but rather makes the different pieces of the problem arguments of the function which starts the execution; this is obviously not a very satisfying solution for a number of reasons. Autognostic does, however, have a fair account of the content for such a specification: a task plus an initial knowledge state (*i.e.*, a set of inputs). The precise details may be somewhat more complex for TMKL, however, both because the existence of multiple domains complicates the specification of the initial knowledge and because we don't currently have language constructs for specifying values for arbitrary concepts and relations (a very big problem). Obviously, we would like our mechanism for specifying these problems to be syntactically consistent with our modeling language.

The second mechanism we need is one for telling a system to make a specific change to a model (*e.g.*, "add to the Mosaic system the capability to allow users to configure their own external browsers"). This is something that Autognostic does not do (it only makes changes to a model in response to a failed execution, not in response to a request from the user to add some new functionality). We do not yet have an account of how to specify such problems. Again, though we would want syntactic consistency with the modeling language.

Both of these issues (especially the latter) seem like issues to be resolved in the future, after we have at least a coherent "first cut" at the modeling language itself.

#5.2 Predicates

I am still troubled by the nature of the predicates (*a.k.a.*, domain relations, relations, etc.) in the system. One of the issues which TMKL doesn't address is the relationship between predicates whose truth is inherent in the values of their arguments and those whose truth may vary. For example, consider the propositions (on block-A block-B) and (equal number-A number-B). It seems potentially reasonable to say that a block stacking system that makes the former proposition true have the "on" predicate as an output, *i.e.*, that it is changing the value of this predicate. In contrast, it seems like it is not really meaningful to say that the "equal" predicate changes in some way. This issue is tightly connected to #5.1 since we will need a much deeper understanding of predicates in order to be able to specify their initial values.

Note that to some extent there is a symmetry between the definition of the internal mechanisms of a predicate and those of a procedure (*i.e.*, a primitive task). Both a predicate and a procedure relate values to each other, the latter simply has a more explicit notion of which values are "input" and which are "output." For more discussion on the issue of procedures, see #5.8.

#5.3 Functions and sub-functions

A key element in the specification of a task is the identification of what comes into the task and what goes out of the task. In TMK, the ":given", ":makes", ":under-condition", and ":semantics" slots specify this information. These slots roughly correspond to the ZD ":if" and ":tomake" slots. However, ZD also has a separate mechanism, the ":provided" and ":addres" slots which describe incidental requirements and results. Ashok Goel (personal communication) has proposed a mechanism for handling this sort of information using a sort of sub-function mechanism referred to as a "side-effect" slot. For example if we take a version of export-comma service that requires 2 digit year data and produces 4 digit year data, we can write the input and output relations using the ZD content:

```
(deftask export-comma
 ...
 :if (loaded timecard data)
 :tomake (in-form comma data)
 :provided (lessp (year-precision data) 100)
 :addres (lessp (year-precision data) 10000)
 ...
```

but we can also convert this to the proposed sub-task format:

```
(deftask export-comma
 ...
 :if (loaded timecard data)
 :tomake (in-form comma data)
 :side-effect export-comma-year-compliance
 ...

(deftask export-comma-year-compliance
 ...
 :if (lessp (year-precision data) 100)
 :tomake (lessp (year-precision data) 10000)
 ...
```

At first glance, this approach seems more complicated than the ZD approach and appears to be effectively equivalent in terms of what sort of inferences can be drawn. On the other hand, there seems to be a somewhat different theoretical stance being taken here (*i.e.*, that there is a decomposition of functionality which is in some way orthogonal to the decomposition specified by the method which accomplishes the task). For this version of TMKL, I have ignored the ":provided" / ":address" / ":side-effect" issue altogether and have used only ":given" and ":makes" (corresponding to ZD's ":if" and ":tomake").

#5.4 Changing values

One issue that I have been moderately concerned about is the issue of addressing the different values before and after a task. This is related to the issue of variable vs. constant values addressed in #2.4 and indirectly in #3.3 but is a separate issue. The concern here is that sometimes our assertions about the results of a task or method should depend on the values which were present before the task or method was executed. For example, say we have a task whose job is to find the "next" case out of a set of cases (for some potentially complex ordering). We might want to say something like this:

```
(deftask find-next-case
 :domain case-domain,
 :input (current-case case-memory)
 :output (current-case)
 :given (not (last current-case case-memory))
 :makes (after current-case #current-case))
```

Here I use the syntax "#current-case" to refer to the value that current-case had before the task was executed; the # syntax comes from ZD which, I believe, means something slightly different than this use. At first glance, this issue appears to be a consequence of the decision to make values variable (from #2.4) but I don't think this is the case. In fact, the same issue can arise with variable relations. Consider, for example, a service which converts a timecard between tab and comma format, either way. This could be encoded:

```
(defservice toggle-format
 :if (loaded system data)
 :tomake (or (and (#in-form tab data) (in-form comma data))
             (and (#in-form comma data) (in-form tab data)))
 )
```

This statement says that either the data was in tab form and is now in comma form *or* the data was in comma form and is now in tab form. In practice we would probably wouldn't want to do this; instead we would want to write this particular unit as two separate services (tab-to-comma and comma-to-tab) which would obviate the issue (since we could put the prior conditions in the :given slots). In general however, this is something we might want to express; my experience and intuition suggests that it is something which comes up much more often with variable values than with variable predicates.

If we decide that we want to allow this sort of capability, the question of what syntactic mechanism to use to do so arises. One approach would be to use a syntax similar to the existing operators. For example the "(after current-case #current-case)" proposition could be written "(after current-case (prior current-case))". The problem that I have with this definition is that this functionality is inherently very different from the functionality of an operator. In particular, the result of the prior "operator" depends on the *variable* involved in the relation, not just the *value* of that variable. For example, under the current rules, the expression:

```
  (and (equal foo bar) (P (S foo)) (not (P (S bar))))
```

can, by definition, never be true. If we allow S to be the prior "operator", however, this no longer holds. Because this capability is inherently different from operators, I think that a distinctive syntax is needed. I like the "#" syntax which I have used above; it is simple and seems reasonably clear.

This issue did not arise in the example developed for TMKL so I have left it open. It may be that we feel that the issue is either rare enough in practice or easy enough to work around that we don't want to deal with it (and complicate both our language and our inferencing rules).

#5.5 Prototypes

The TMK language supports a concept known as "prototypical tasks." As I understand it, the basic idea behind this concept is that there are some tasks which are instantiations of some more general class of tasks. To some extent the notion is like that of Generic Tasks [Chandrasekaran, 1988], except that instead of having an a priori set of extremely general tasks like classification we have more precise prototypes that are specific to certain classes of tasks. For example, the REFLECS system has a prototypical obstacle avoidance task of which various specific obstacle avoidance strategies are instances. In practice, however, the task prototyping mechanism in Autognostic was largely (as far as I have been able to determine) used as a construct for implementing loops (because, as I understand it, the trace facility is able to store / analyze traces more efficiently when loop bodies were implemented as prototypical tasks).

In general, it seems like the issue of task prototypes becomes considerably more important if we are implementing multiple reasoning systems within a single domain. I think that one reason that Autognostic's account of prototypical tasks is severely limited is that no examples have been done which combine multiple systems. The issue of task prototypes is an important one; I have not addressed it in the current language but intend to do so eventually.

#5.6 Correspondence details

In #2.2, it is argued that tasks and methods need to be able to select which correspondences are used to link them to other tasks and methods in cases where more than one correspondence between their domains has been defined. No examples of this phenomenon occur in the example (no two domains in the example have more than one mapping between them). Because it has not come up, I have not yet constructed a mechanism for this in TMKL. In general this could be a marginally tricky issue because we have to connect tasks not only to the higher level methods they support and the lower level methods that implement them but also to the other tasks at the same level that they take input from or provide output to. However, I don't think there are any deep theoretical concerns here (aside from those already raised in #2.2).

#5.7 Knowledge design parameters

In #2.4, I state that TMKL allows both predicates and concepts to be altered by tasks and methods in the system. While this adds considerable power, it also potentially compromises one of Autognostic's primary tools for making design changes; Autognostic considers the truth values of relations to be design parameters of the system being modeled and some of its redesign strategies revolve around changing these relations.

In general, the values of both concepts *and* relations might be useful parameters of the design of the system itself. Perhaps the model should have a mechanism for specifying default initial values for both of these types of data. In this way we would get the redesign power of being able to make changes to system values as a redesign strategy but also preserve the execution power of being able to modify arbitrary values. The language for specifying these initial defaults would presumably be compatible with the language for specifying input values discussed in #5.1 (which also has to deal with the issue of specifying values).

An important point to note here is that we will need some account of "hardness" of various design parameters to control which sorts of things we attempt to redesign first, *i.e.*, we want to attempt to redesign more "soft" aspects of the design before attempting to redesign more "hard" aspects. For example, we might say that concept values are more hard than relation values. Autognostic has a variety of such heuristics for ordering selection of repair strategies. As we work though the reasoning issues involved in using TMKL, it is likely that we will develop more.

#5.8 Procedures

One issue which arises in TMK and TMKL but not in ZD is the definition of procedures, *i.e.*, the implementation of bottom level, primitive tasks. Because TMK and TMKL models are intended to actually be "run," their behavior needs to be specified at the level of an actual working system. TMK handles this by making procedures be arbitrary LISP functions. This works, but introduces some serious limitations (since one can't really reason about the internals of the functions themselves). One option is to make all primitives involve basic logical operations (*e.g.*, assert this statement to be true, etc.). This would allow the system to reason across all levels; however, it is likely to be thoroughly intractable both to reason about such models and to build them in the first place. Another option is to encode procedures as explicit lists of mappings between input and output values; this is relatively simple to specify and easy to manipulate but is generally not powerful enough for all of the kinds of procedures we might want to implement.

One compromise on this issue would be to provide our own language for specifying procedures which is more powerful than simple logical operations but more constrained than arbitrary LISP functions. However, if we made such a language too powerful it would constitute an entirely new general purpose programming language and an associated interpreter (which is obviously an enormous problem in and of itself).

Another possible compromise would be to provide multiple mechanisms for specifying procedures. Those procedures which are simple enough to be described as logical operations or value mappings could be (thus enabling the kinds of reasoning that these representations support). Conversely, those which are sufficiently complex as to require restricted or even unrestricted LISP code may be represented as such with a corresponding loss in the ability to reason about the procedure.

The TMKL example provided in the appendix does not represent the primitive procedures at all so there is no resolution of this issue here.

#6 Research Agendas

This document concludes with a description of a few potential avenues for further research involving models of the sort proposed here. There are a great many potential perspectives on the modeling of computation and a great many potential benefits. I am presenting only a sample of them here:

#6.1 Proactive redesign

The use of TMK in SIRRINE and Autognostic is largely focused on failure-driven redesign; a system fails to accomplish some specified task and thus makes modifications which will avoid this failure in the future. Such failures may arise either from flaws in the original design or from changes in the context of operation. However, some contextual changes may be sufficiently abrupt or complex that the environment may simply not provide enough information to determine the appropriate new functionality. In some cases, a user may know what the new functionality should be. For example, consider the use of a meeting scheduling system in an environment in which a new policy is announced that no meetings may be scheduled on Monday for any reason (presumably to provide one day a week in which people can work without interruptions). In the existing SIRRINE and Autognostic systems, the only way for a user to have the meeting scheduler learn this new policy is to reject every meeting it suggests on Mondays until (hopefully) the system eventually learns not to schedule meetings on Monday. However, in this scenario it is far more reasonable for the user to simply tell the system what the new policy is rather than correct the system every time it fails to obey the new policy.

If new requirements were specified in advance rather than simply being implicit in post hoc feedback, a system would effectively performing *partially* automated self-adaptation; the user provides high level guidance as to what new capabilities the system should have and the system decides how to achieve those capabilities. TMKL could potentially provide both the ontology for specifying new functional requirements (as discussed in #5.1) and the mechanisms for performing such adaptations.

#6.2 Patterns of redesign

Autognostic has a set of redesign strategies which it uses to adapt TMK models. One important research objective is the development of new redesign strategies to provide an increasingly broad range of adaptation capabilities. Most of the other research goals described here will probably require development of new redesign strategies to support them. For example, the proactive design task described in #6.1 will probably require new strategies capable of doing redesign to meet more elaborate specifications of the desired results.

Furthermore, it would be extremely desirable for a system to be able to *learn* new redesign strategies while in use. The redesign strategies in Autognostic are "hard-coded" much like the redesign strategies in the Kritik [Goel et al., 1997] system (which redesigns physical devices). One of the ways in which IDeAL [Bhatta, 1995] extends the Kritik paradigm is by supporting flexible learning of new redesign strategies. An important research goal is an analogous extension to the Autognostic / TMK paradigm.

#6.3 Meta-reflection

What benefits can be gained by allowing the mechanism for reflection to reflect upon themselves? One of these might be a principled account of the learning of new repair strategies (as discussed in #6.2). In fact, any of the different applications of TMKL models could potentially be enhanced by representing those applications in TMKL and allowing a reflective learning agent to reflect on the process of reflection itself. This issue clearly merits further investigation.

#6.4 Generic computational elements

The idea of prototypical tasks (discussed in #5.5) suggests a larger research topic: the notion that some elements of computation (*e.g.*, tasks, methods, knowledge) may be generic, *i.e.*, may exist across a variety of reasoning goals in a variety of domains. For example, one could argue that there is a generic case-based reasoning method which sets up the subtasks of retrieval, adaptation, verification, and storage. For specific tasks, this generic method is realized by specific kinds of subtasks. For example, an instance of the generic case-based reasoning method would be a case-based design method which specified a redesign task as its adaptation subtask and further realized that redesign task using a model-based repair method. One could represent the commonalities between case-based reasoning methods in multiple domains by declaring these methods to be instances of a prototypical method. However, in general we would want to say not only *that* these methods shared common traits but also *what* those common traits were. In other words, the prototypical case-based reasoning method should abstractly describe the case-based reasoning process. One obvious example of the use of such a generic structure would be in design; one could design a system to do case-based reasoning by simply instantiating the abstract generic mechanism. Another use of such a mechanism would be in guiding analogical transfer; by identifying which structures are instances of the same prototypical structure, a system could form analogies between different fully realized models.

Can we provide a powerful, broadly applicable taxonomy of generic tasks, methods, and knowledge? What are the different benefits which can be obtained from such a taxonomy? How can we represent these generic structures in such a way to derive these benefits. These remain open research questions.

#6.5 Situated agents

Consider two varieties of primitive tasks: primitive tasks which take place entirely in the mind and those which involve perception and / or action in the world. TMK does not make an explicit distinction between these two kinds of primitive actions; this largely emerges from the problem solving view discussed in #3.3. A task-method analysis of cognition seems to have the potential to provide very effective accounts of reasoning which combines both internal processing and external interaction because it can build higher level tasks and methods from primitive elements of both varieties (as in REFLECS). However, I think that more care is needed to address some of the issues raised by the distinction between mental and embodied actions.

As an example of such an issue, consider the representation of *environmental variables*. In [Kirlik, 1998], *q.v.*, environmental variables are characterized by two properties: accessibility to perception and accessibility to action. Accessibility in this context is defined as either proximal (things which can be seen or touched) or distal (things which can only be perceived or affected indirectly). Thus there are potentially four kinds of environmental variables in this framework (proximal perception + proximal action, proximal perception + distal action, distal perception + proximal action, and distal perception + distal action). To the extent that we can view them in this way, I would argue mental variables are inherently proximal perception + proximal action (since they are present in the mind). One way to address this distinction in TMKL is to define proximal variables as those for which primitive tasks exist which perceive or act upon them directly and distal ones as which only interact with non-primitive tasks. Note that this requires that we have a mechanism for interacting with variables at higher levels which cannot be interacted with at lower levels. TMK does not have such a mechanism; the only way to affect or make use of a variable in a high level task is to have it decompose into one or more subtasks which affect or use that variable. The interactions between domains / clusters in ZD might provide a first step toward such an account but I don't believe it would be a complete account, since the high level variables still have to be directly mapped to lower level ones. I suspect that a thorough account might involve some sort of mechanism whereby distal variables can be viewed as combinations or aggregations of mental and / or proximal environmental variables.

The proximal / distal distinction is just one of many issues that need to be resolved in building task-method models of agents in situated task domains. An effective task-method theory of both reasoning *and* acting requires further research.

#6.6 Distributed cognition / multi-agent societies

There are a great many reasons why agents in a multi-agent society would need to have models of their own processing. One of these is that societies of agents are typically dynamic and thus an agent needs to be able to adapt itself to the changing demands of the society. This capability is essentially implicit in the failure-driven self-redesign paradigm found in Autognostic and SIRRINE. However there are a great many other applications of reflective self-knowledge in these sorts of environments. For example, effective, flexible cooperation between agents can clearly be facilitated if both agents know what it is they do and can communicate this knowledge to each other. The primary goal of research in distributed intelligent systems (as I understand it) is the development of an environment in which an agent which requires some functionality that it does not possess simply announces what it needs done and some other agent "volunteers" to do it. It is clear that to fully realize these goals, agents will need to know what it is they do and don't do. TMKL could potentially provide an account of such knowledge.

Another example of an application of self-knowledge in multi-agent societies is the idea of absorption of the functionality of external agents. Under some conditions, it may be possible to inductively infer a task-method model of an external agent and consequently to duplicate the functionality of that agent. For example, if a system relies on an external file management agent for storage and retrieval of data, that system might be able to observe the changes to the file structure which take place when the file management agent is invoked and infer a description of what that agent does and how it does it. If this descriptions becomes sufficiently precise with a sufficiently high confidence, it may be possible for the system to act as its own file management agent. This would be particularly useful if the use of the file management agent incurs a significant cost or that agent suddenly becomes unavailable.

The ability to infer, even partially, what tasks, methods, and knowledge are used by another agent could provide a great many other capabilities. For example, consider environments in which agents compete with each other (*e.g.*, games, economic environments, military engagements). Search-based AI systems have done very well in many such environments for which full world knowledge is available (*e.g.*, chess). AI has been much less successful in environments where knowledge of the state of the world is very limited (*e.g.*, poker). In such environments, successful resolution often depends more on understanding and predicting the strategies of the opponent(s) than on the development of one's own strategy. For agents to effectively compete in these limited information domains, I claim that they need mechanisms for understanding *what* other agents do and *how* they do so.

It is important to note that these issues relating to multi-agent societies are tightly linked to the problem of integrating reasoning and action discussed in #6.5. Communicating with external agents is clearly an action rather than an internal computation and needs to be considered as such to effectively model social agents.

Discussion

What exactly is TMKL? What does it mean to have a model of a computation in this language being proposed here. Three key perspectives on what TMKL is encoding are:

- How an agent[3] works. The claim that this perspective makes is that reasoning is done by selecting methods to address tasks and instantiating subtasks to accomplish methods. This is essentially the Router [Goel et al., 1994] perspective. In this sense, TMKL can be viewed as analogous to (or in competition with) cognitive architectures such as ACT-R [Anderson, 1993], SOAR [Laird et al., 1987], blackboard architectures [Hayes-Roth & Hayes-Roth, 1979], etc.

- What an agent knows about itself. The claim that this perspective makes is that agents know about the tasks, methods, and knowledge they use (and further utilize this knowledge in adapting themselves). This is essentially the TMK / Autognostic view. In this sense, TMKL can be viewed as analogous to AI theories of reflection / meta-reasoning such as MOLGEN [Stefik, 1981], Theo [Mitchell et al., 1989], Meta-Aqua [Cox, 1996], etc.

- What a designer knows about a system being designed. The claim that this perspective makes is that descriptions of tasks, methods, and knowledge facilitate redesign of systems. This view has been advanced in the MORALE project [Abowd et al., 1997] and seems essential to ZD as

[3] Note that I am deliberately avoiding the issue here of whether this is a theory of how human agents *are* constructed or whether this is a theory of how AI systems *can be* constructed. I assume that a substantive contribution to either of these constitutes at least a potentially valuable insight into the other. This assumption has been analyzed in enormous depth by virtually everyone who has studied cognitive science and strong arguments exist on either side; a detailed analysis of this issue is far beyond the scope of this report.

well. In this sense, TMKL can be viewed as analogous to architectural description languages such as those described in [Clements, 1996].

My working hypothesis is that these three perspectives are intimately interconnected. For example, to the extent that self-models are accurate, the first and second perspectives above are indistinguishable. Similarly, to the extent that adaptive learning can be viewed as a redesign process, the second and third perspectives also seem largely isomorphic. Are all three of these perspectives sufficiently similar that they can be addressed by a single language? Further development and instantiation of the proposed TMKL framework should provide empirical insight into this question.

Acknowledgments

Substantial technical contributions to this document have been made by Dean Allemang, Ashok Goel, Spencer Rugaber, and the Georgia Tech MORALE research group. Furthermore, the proposals in this paper build upon work done on the Autognostic project and the ZD project and is thus are strongly dependent on the accomplishments of those involved in these projects, including Eleni Stroulia, Ashok Goel, and Dean Allemang.

Effort sponsored by the Defense Advanced Research Projects Agency, and Rome 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 author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[Abowd et al., 1997] Abowd, G., Goel, A. K., Jerding, D. F., McCracken, M., Moore, M., Murdock, J. W., Potts, C., Rugaber, S., & Wills, L. (1997). MORALE – Mission oriented architectural legacy evolution. In *Proceedings International Conference on Software Maintenance 97* Bari, Italy.

[Allemang, 1997] Allemang, D. (1997). ZD tutorial. Unpublished.

[Anderson, 1993] Anderson, J. R. (1993). *Rules of the Mind.* Hillsdale, NJ: Lawrence Erlbaum Associates.

[Bhatta, 1995] Bhatta, S. (1995). *Model-Based Analogy in Innovative Device Design.* PhD dissertation, Georgia Institute of Technology, College of Computing.

[Chandrasekaran, 1988] Chandrasekaran, B. (1988). Generic tasks as building blocks for knowledge-based systems: The diagnosis and routine design examples. *Knowledge Engineering Review*, 3(3), 183–219.

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

[Cox, 1996] Cox, M. T. (1996). *Introspective multistrategy learning: Constructing a learning strategy under reasoning failure.* PhD dissertation, Georgia Institute of Technology, College of Computing.

[Goel et al., 1994] Goel, A., Ali, K., Donnellan, M., Gomez, A., & Callantine, T. (1994). Multistrategy adaptive navigational path planning. *IEEE Expert*, 9(6), 57–65.

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

[Goel et al., 1996] Goel, A. K., Gomez, A., Grue, N., Murdock, J. W., Recker, M., & Govindaraj, T. (1996). Explanatory interface in interactive design environments. In J. S. Gero & F. Sudweeks (Eds.), *Proceedings of the Fourth International Conference on Artificial Intelligence in Design* Stanford, California: Kluwer Academic Publishers.

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

[Griffith, 1997] Griffith, T. W. (1997). *A Computational Theory of Dynamical Modeling in Scientific Reasoning*. PhD thesis proposal, Georgia Institute of Technology, Cognitive Science Program, College of Computing.

[Hayes-Roth & Hayes-Roth, 1979] Hayes-Roth, B. & Hayes-Roth, F. (1979). A cognitive model of planning. *Cognitive Science*, 3, 275–310.

[Kirlik, 1998] Kirlik, A. (1998). The ecological expert: Acting to create information to guide action. In *Proceedings of the 1998 Conference on Human Interaction with Complex Systems, HICS-98* Dayton, OH.

[Laird et al., 1987] Laird, J., Newell, A., & Rosenbloom, P. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33.

[Liver & Allemang, 1985] Liver, B. & Allemang, D. T. (1985). A functional representation for software reuse and design. *International Journal of Software Engineering and Knowledge Engineering*, 5(2), 227–269.

[Mitchell et al., 1989] Mitchell, T. M., Allen, J., Chalasani, P., Cheng, J., Etzioni, O., Ringuette, M., & Schlimmer, J. C. (1989). Theo: A framework for self-improving systems. In K. VanLehn (Ed.), *Architectures for Intelligence*. Erlbaum.

[Stefik, 1981] Stefik, M. (1981). Planning and meta-planning (MOLGEN: Part 2). *Artificial Intelligence*, 16(2).

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

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

[Stroulia & Goel, 1996] Stroulia, E. & Goel, A. (1996). A model-based approach to blame assignment: Revising the reasoning steps of problem solvers. In *Proc. National Conference on Artificial Intelligence - AAAI96* Portland, Oregon.

[Stroulia & Goel, 1994] Stroulia, E. & Goel, A. K. (1994). A model-based approach to reflective learning. In F. Bergadano & L. D. Raedt (Eds.), *Proc. 1994 European Conference on Machine Learning* (pp. 287–306). Catania, Italy.

Appendices

As discussed at the beginning of this document, I have been using three different models of the IOP system as a driving factor for this analysis. The three models encode this system in ZD, TMK, and TMKL respectively. Diagrams which provide an overview of these three models appear hear as Figures 1, 2, and 3 respectively.

Appendix A: zd.lisp

```
; zd.lisp
; this is the IOP example, transcribed from the tutorial
; Whenever alternates were presented, this file containts the final,
;  working alternates.  Furthermore, some typos have been fixed
;  reflecting email correspondence

; Revised 11/9/97
; The comment "***" below indicates lines which differ from the
;  "official" current version (except in spacing and capitalization
;  which differ throughout).  Most of the differences simply involve
;  the use of "PREDICATE" in the place of "FUN" and the use of "NQTHM"
;  in the place of "NQ".

(defSDL NQTHM NQTHM) ; ***

(defcluster service IOP
  :sdl NQTHM
  :declare ((ETC :type CONS)
    (LOADED :type (PREDICATE 2))
    (TS-TAM :type NUMBER)
    (TS-E :type NUMBER))
  )

(defservice submit
  :If (LOADED TS-E ETC)
  :ToMake (LOADED TS-TAM ETC)
  )

(add-to-cluster submit IOP)

(defcluster implementation submission
  :sdl NQTHM ; ***
  :use (IOP)
  :declare ((M-TAM :type NUMBER)
    (M-E :type NUMBER)
```

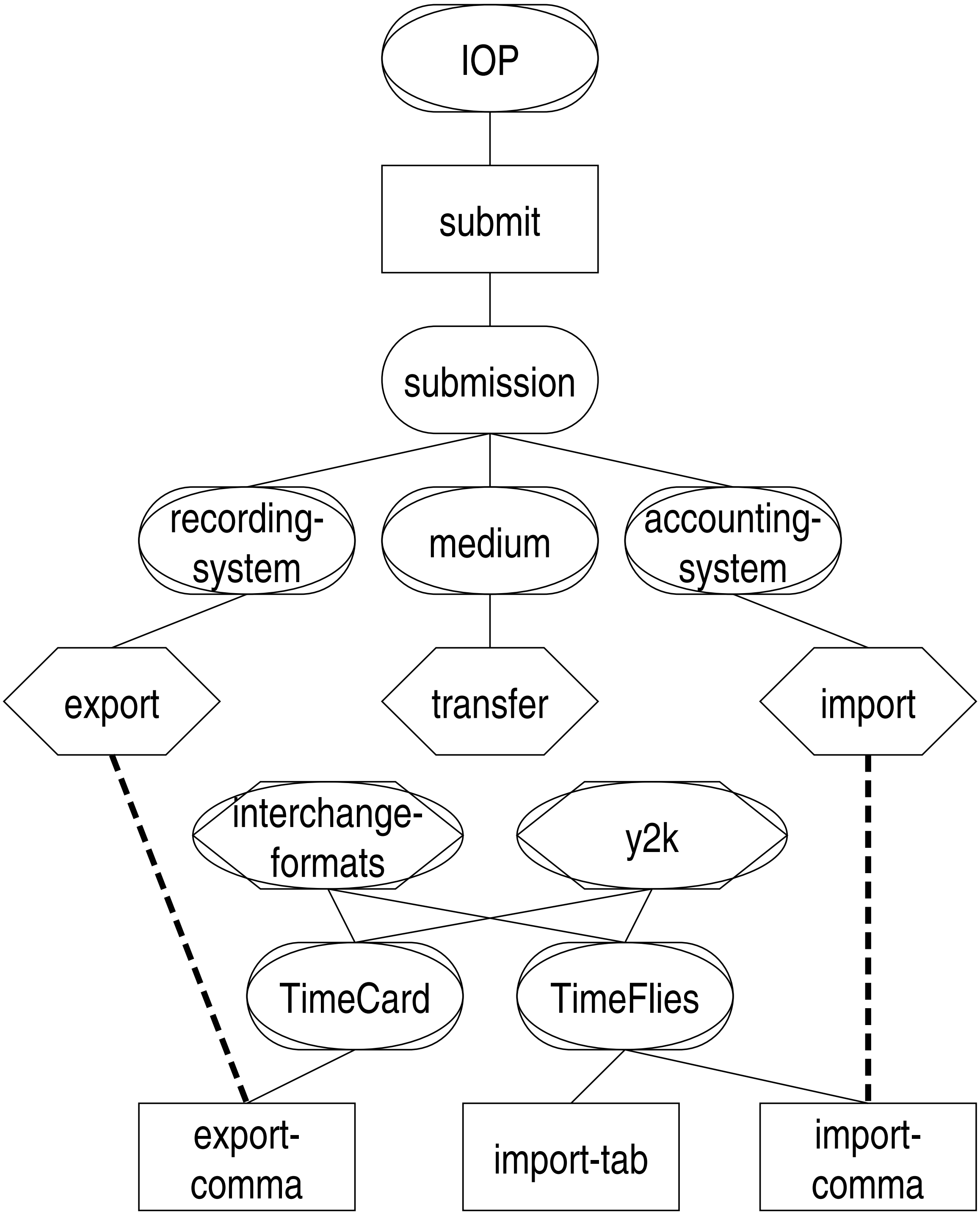

Figure 1: The ZD model of the IOP system from zd.lisp

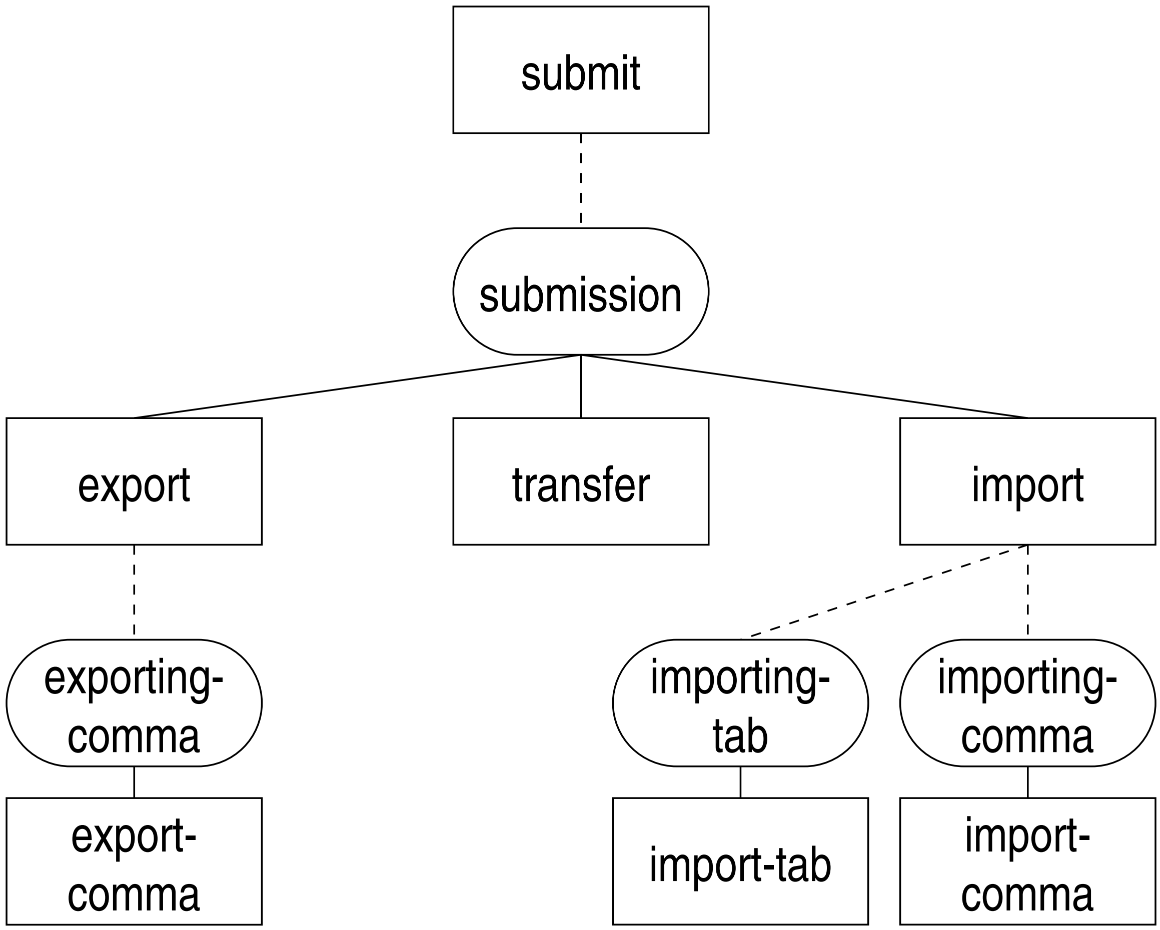

Figure 2: The TMK model of the IOP system from tmk.lisp

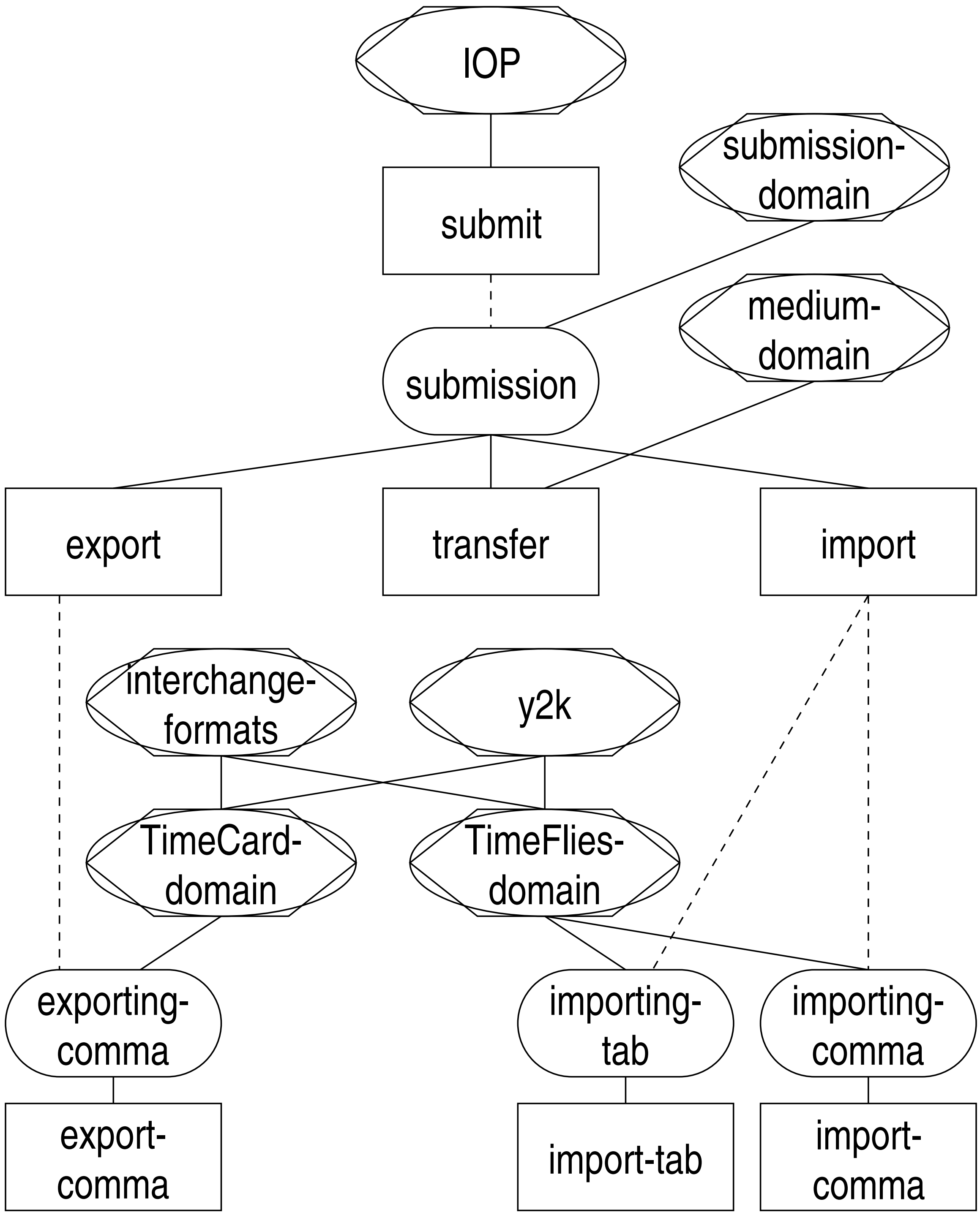

Figure 3: The TMKL of the IOP system from tmkl.lisp

```
    (PRESENT :type (PREDICATE 2)) ; ***
    (in-form :type (PREDICATE 2)) ; ***
    (format :type NUMBER))
  )

(defcluster requirement recording-system
  :sdl NQTHM ; ***
  :use (submission)
  :declare ()
  )

(defrequirement export
  :If (LOADED TS-E ETC)
  :ToMake (and (in-form format ETC) (PRESENT M-E ETC))
  )

(add-to-cluster export recording-system)

(defcluster requirement accounting-system
  :sdl NQTHM
  :use (submission)
  :declare ()
  )

(defrequirement import
  :If (and (in-form format ETC) (PRESENT M-TAM ETC))
  :ToMake (LOADED TS-TAM ETC)
  )

(add-to-cluster import accounting-system)

(defcluster requirement medium
  :sdl NQTHM ; ***
  :declare ((PRESENT :type (PREDICATE 2)) ; ***
    (source :type NUMBER)
    (destination :type NUMBER)
    (message :type CONS))

(defrequirement transfer
  :If (PRESENT source message)
  :ToMake (PRESENT destination message)
  )

(add-to-cluster transfer medium)

(defrelation recording-system submission)

(defrelation accounting-system submission)
```

```
(defrelation medium submission
  :representation ((source M-E)
   (destination M-TAM)
   (message ETC))
  )

(defimpl submit
  :nodeforms
  ((n0 (LOADED TS-E ETC))
   (n1 (and (in-form format ETC) (PRESENT M-E ETC)))
   (n2 (and (in-form format ETC) (PRESENT M-TAM ETC)))
   (n3 (LOADED TS-TAM ETC)))
  :transforms
  ((n0 n1 single recording-system.export)
   (n1 n2 single medium.transfer)
   (n2 n3 single accounting-system.import))
  )

(add-to-cluster submit submission)
(defrelation submission IOP)

(defdomain interchange-formats
  :SDL NQTHM
  :types ((form :isa number))
  :declare ((comma :type form)
    (tab :type form))
  :constraints ((not (equal comma tab)))
  )

(defdomain Y2K
  :SDL NQTHM ; ***
  :declare (year-precision (fun 1)) ; ***
  )

(defcluster service TimeCard
  :SDL NQTHM ; ***
  :use (interchange-formats Y2K)
  :declare ((loaded :type (PREDICATE 2))
    (timecard :type number)
    (in-form :type (PREDICATE 2)) ; ***
    (data :type cons)
    )
  )

(defservice export-comma
  :if (loaded timecard data)
  :tomake (in-form comma data)
```

```
  :addres (lessp (year-precision data) 10000)
  )

; ***
; service "export-tab" would have gone here.

(add-to-cluster export-comma TimeCard)
; ***
; add-to-cluster "export-tab" would have gone here

(defcluster service TimeFlies
  :SDL NQTHM
  :use (interchange-formats Y2K)
  :declare ((loaded :type (PREDICATE 2))
    (timeflies :type number)
    (data :type CONS)
    (in-form :type (PREDICATE 2)) ; ***
    )
  )


(defservice import-comma
  :if (in-form comma data)
  :tomake (loaded timeflies data)
  :provided (lessp (year-precision data) 100)
  )

(add-to-cluster import-comma TimeFlies)

(defservice import-tab
  :if (in-form tab data)
  :tomake (loaded timeflies data)
  )

(add-to-cluster import-tab TimeFlies)

(defglue TimeCard recording-system
  :terms ((loaded loaded)
  (timecard ts-e)
  (in-form in-form)
  (comma format)
  (data etc))
  :funs ((export-comma export))
  )

(defglue TimeFlies accounting-system
  :terms ((loaded loaded)
  (timeflies ts-tam)
```

```
  (in-form in-form)
  (comma format)
  (data etc))
  :funs ((import-comma import))
  )
```

Appendix B: tmk.lisp

```
(defclass d-card (domain-concept)
  ()
  (:default-initargs
   :name 'd-card
   :identity-test 'equal
   :attributes (list
(make-instance 'attribute
       :name 'location
       :function 'car
(make-instance 'attribute
       :name 'car
       :function 'cadr)
(make-instance 'attribute
       :name 'cdr
       :function 'cddr)))
   ))

(defclass d-number (domain-concept)
  ()
  (:default-initargs
   :name 'd-number
   :identity-test '=
   ))

(defclass d-TS (d-number)
  ()
  (:default-initargs
   :name 'd-TS
  ))

(defclass d-M (d-number)
  ()
  (:default-initargs
   :name 'd-M
  ))

(defclass d-format (d-number)
  ()
  (:default-initargs
   :name 'd-format
```

```
   :domain 'format-domain
  ))

(defclass ETC (d-card problem-variable)
  ()
  (:default-initargs
   :name 'ETC
   :input-to (find-where-info-is-input known-tasks 'ETC)
   :produced-by (find-where-info-is-output known-tasks 'ETC))
  ))

(defclass TS-TAM (d-TS problem-variable)
  ()
  (:default-initargs
   :name 'TS-TAM
   :input-to (find-where-info-is-input known-tasks 'TS-TAM)
   :produced-by (find-where-info-is-output known-tasks 'TS-TAM))
  ))

(defclass TS-E (d-TS problem-variable)
  ()
  (:default-initargs
   :name 'TS-E
   :input-to (find-where-info-is-input known-tasks 'TS-E)
   :produced-by (find-where-info-is-output known-tasks 'TS-E))
  ))

(defclass M-TAM (d-M problem-variable)
  ()
  (:default-initargs
   :name 'M-E
   :input-to (find-where-info-is-input known-tasks 'M-E)
   :produced-by (find-where-info-is-output known-tasks 'M-E))
  )

(defclass M-E (d-M problem-variable)
  ()
  (:default-initargs
   :name 'M-E
   :input-to (find-where-info-is-input known-tasks 'M-E)
   :produced-by (find-where-info-is-output known-tasks 'M-E))
  ))

(defclass format (d-format problem-variable)
  ()
  (:default-initargs
   :name 'format
   :input-to (find-where-info-is-input known-tasks 'format)
```

```
   :produced-by (find-where-info-is-output known-tasks 'format))
  ))

(setf IOP-domain-objects
      '(d-card d-number d-TS d-M d-format d-form))

(setf format-domain
      (list (setf COMMA 1) (setf TAB 2)))

(setf IOP-domain-relations
  (list
   (setf rel00 (make-instance 'domain-relation
      :name 'LOADED
      :organizational nil
      :input-arg-types '(d-card d-TS)
      :output-arg-types '()
      :data-structure 'LOADED-relation))
   (setf rel01 (make-instance 'domain-relation
      :name 'PRESENT
      :organizational nil
      :input-arg-types '(d-card d-M)
      :output-arg-types '()
      :data-structure 'PRESENT-relation))
   (setf rel02 (make-instance 'domain-relation
      :name 'in-form
      :organizational nil
      :input-arg-types '(d-card d-format)
      :output-arg-types '()
      :data-structure 'in-form-relation))))


; Tasks are organized as a list:
; :name :input :output :by :prototype :procedure :semantics :under-condition

; MMethods are organized as a list:
; :name :subtasks :control :under-condition

(setf IOP-ts
        '(
        ---task
        submit ; :name
        (TS-TAM TS-E ETC M-TAM M-E format) ; :input
        (ETC) ; :output
        (submission) ; :by
        nil ; :prototype
        nil ; :procedure
        ((LOADED TS-TAM nil ETC nil)) ; :semantics
        ((LOADED TS-E nil ETC nil)) ; :under-condition
```

```
---mmethod
submission ; :name
(export transfer import) ; :subtasks
((make-instance 'serial-op) export transfer import) ; :control
((LOADED TS-E nil ETC nil)) ; :under-condition

---task
export ; :name
(TS-E ETC M-E format) ; input
(ETC) ; output
(exporting-comma) ; :by
        nil ; :prototype
        nil ; :procedure
        ((in-form format nil ETC nil)
 (PRESENT M-E nil ETC nil)) ; :semantics
((LOADED TS-E nil ETC nil)) ; :under-condition

---method
exporting-comma ; :name
(export-comma) ; :subtasks
((make-instance 'serial-op) export-comma) ; :control
((LOADED TS-E nil ETC nil)) ; :under-condition

---task
export-comma ; :name
(TS-E ETC M-E format) ; input
(ETC) ; output
nil ; :by
nil ; :prototype
do-export-comma ; procedure
((in-form COMMA nil ETC nil)
 (#'lessp (year-precision data) nil 10000 nil)) ; :semantics
((LOADED TS-E nil ETC nil)) ; :under-condition

---task
transfer ; :name
(M-E M-TAM ETC) ; input
(ETC) ; output
nil ; :by
nil ; :prototype
do-transfer ; procedure
((PRESENT M-E nil ETC nil)) ; :semantics
((PRESENT M-TAM nil ETC nil)) ; :under-condition

---task
import ; :name
(TS-TAM ETC M-TAM format) ; input
```

```
(ETC) ; output
(importing-comma importing-tab) ; :by
        nil ; :prototype
        nil ; :procedure
((LOADED TS-TAM nil ETC nil)) ; :semantics
        ((in-form format nil ETC nil)
 (PRESENT M-TAM nil ETC nil)) ; under-condition

---method
importing-comma ; :name
(import-comma) ; :subtasks
((make-instance 'serial-op) import-comma) ; :control
        ((in-form format nil COMMA nil)
 (#'lessp (year-precision ETC) nil 100 nil)) ; under-condition

---task
import-comma
(TS-TAM ETC M-TAM format) ; input
(ETC) ; output
nil ; :by
nil ; :prototype
do-import-comma ; :procedure
((LOADED TS-TAM nil ETC nil)) ; :semantics
        ((in-form format nil COMMA nil)
 (#'lessp (year-precision ETC) nil 100 nil)) ; under-condition

---method
importing-tab ; :name
(import-tab) ; :subtasks
((make-instance 'serial-op) import-tab) ; :control
        ((in-form format nil TAB nil)
 (#'lessp (year-precision ETC) nil 100 nil)) ; under-condition

---task
import-tab
(TS-TAM ETC M-TAM format) ; input
(ETC) ; output
nil ; :by
nil ; :prototype
do-import-tab ; :procedure
((LOADED TS-TAM nil ETC nil)) ; :semantics
        ((in-form format nil TAB nil)
 (#'lessp (year-precision ETC) nil 100 nil)) ; under-condition
))
```

Appendix C: tmkl.lisp

```
(defdomain IOP
```

```
 :sdl proposed-THM
 :concepts
 (defconcept
   (TS :parent number)
   (card :parent cons
 :identity-test equal))
 :variables
 (defvariable
   (TS-TAM :isa TS)
   (TS-E :isa TS)
   (ETC :isa card)
   (LOADED :isa predicate
   :arg-types (card TS))))

(defdomain submission-domain
 :use (IOP)
 :concepts
 (defconcept
   (M :parent number)
   (format :parent number))
 :variables
 (defvariable
   (M-TAM :isa M)
   (M-E :isa M)
   (format :isa format)
   (PRESENT :isa predicate
    :arg-types (card M))
   (in-form :isa predicate
    :arg-types (card format))))

(defdomain medium-domain
 :sdl proposed-THM
 :concepts
 (defconcept
   (machine :parent number)
   (message :parent cons
    :identity-test equal))
 :variables
 (defvariable
   (source :isa machine)
   (destination :isa machine)
   (content :isa message)
   (PRESENT :isa predicate
    :arg-types (message machine))))

(defdomain interchange-formats
 :sdl proposed-THM
 :concepts
```

```
 (defconcept
   (form :parent number))
 :variables
 (defvariable
   (comma :isa form)
   (tab :isa form))
 :constraints ((not (equal comma tab))))

(defdomain y2k
 :sdl proposed-THM
 :variables Variable
 (defvariable
   (year-precision :parent function
   :arg-types (cons)
   :result-type number)))


(defdomain TimeCard-domain
 :use (interchange-formats Y2K)
 :concepts
 (defconcept
   (system :parent number)
   (machine :parent number)
   (card :parent cons
 :identity-test equal))
 :variables
 (defvariable
   (timecard :isa system)
   (timecard-machine :isa machine)
   (data :isa card)
   (loaded :isa predicate
   :arg-types (card TS))
   (in-form :isa predicate
    :arg-types (card form))))

(defdomain TimeFlies-domain
 :use (interchange-formats Y2K)
 :concepts
 (defconcept
   (system :parent number)
   (machine :parent number)
   (card :parent cons
 :identity-test equal))
 :variables
 (defvariable
   (timeflies :isa system)
   (timeflies-machine :isa machine)
   (data :isa card)
```

```
  (loaded :isa predicate
  :arg-types (card TS))
  (in-form :isa predicate
   :arg-types (card form))))

(defcorrespondence submission-medium
 :domains (submission-domain medium-domain)
 :map ((M machine)
       (card message)
       (M-TAM source)
       (M-E destination)
       (ETC content)))

(defcorrespondence submission-interchange-formats
 :domains (submission-domain interchange-formats)
 :map ((format form)))

(defcorrespondence submission-TimeCard
 :domains (submission-domain TimeCard-domain)
 :map ((TS system)
       (M machine)
       (TS-E timecard)
       (M-E timecard-machine)
       (ETC data)))

(defcorrespondence submission-TimeFlies
 :domains (submission-domain TimeFlies-domain)
 :map ((TS system)
       (M machine)
       (TS-TAM timeflies)
       (M-TAM timeflies-machine)
       (ETC data)))

(deftask submit
 :domain IOP
 :input (TS-TAM TS-E ETC LOADED)
 :output (LOADED)
 :given (LOADED TS-E ETC)
 :makes (LOADED TS-TAM ETC)
 :by (submission))

(defmethod submission
 :domain submission-domain
 :given (LOADED TS-E ETC)
 :makes (LOADED TS-TAM ETC)
 :subtasks (export transfer import)
 :control (series export transfer import))
```

```
(deftask export
 :domain submission-domain
 :input (TS-E M-E ETC LOADED in-form PRESENT)
 :output (in-form format PRESENT)
 :given (LOADED TS-E ETC)
 :makes (and (in-form format ETC) (PRESENT M-E ETC))
 :by (exporting-comma))

(defmethod exporting-comma
 :domain TimeCard-domain
 :given (loaded timecard data)
 :makes (and (in-form comma data)
     (lessp (year-precision data) 10000)
     (present timecard-machine data))
 :subtasks (export-comma))

(deftask export-comma
 :domain TimeCard-domain
 :input (timecard timecard-machine data loaded in-form present)
 :output (in-form format present)
 :given (loaded timecard data)
 :makes (and (in-form comma data)
     (lessp (year-precision data) 10000)
     (present timecard-machine data))
 :procedure do-export-comma)

(deftask transfer
 :domain medium-domain
 :input (PRESENT source destination content)
 :output (PRESENT)
 :given (PRESENT source content)
 :makes (PRESENT destination content)
 :procedure do-transfer)

(deftask import
 :domain submission-domain
 :input (TS-TAM M-TAM ETC LOADED in-form PRESENT)
 :output (LOADED)
 :given (and (in-form format ETC) (PRESENT M-TAM ETC))
 :makes (LOADED TS-TAM ETC)
 :by (importing-comma importing-tab))

(defmethod importing-comma
 :domain TimeFlies-domain
 :given (and (in-form comma data)
     (lessp (year-precision data) 100)
     (present timeflies-machine data))
 :makes (loaded timeflies data)
```

```
 :subtasks (import-comma))

(deftask import-comma
 :domain TimeFlies-domain
 :input (timeflies timeflies-machine data loaded in-form present)
 :output (loaded)
 :given (and (in-form comma data)
     (lessp (year-precision data) 100)
     (present timeflies-machine data))
 :makes (loaded timeflies data)
 :procedure do-import-comma)

(defmethod importing-tab
 :domain TimeFlies-domain
 :given (and (in-form tab data)
     (lessp (year-precision data) 100)
     (present timeflies-machine data))
 :makes (loaded timeflies data)
 :subtasks (import-tab))

(deftask import-tab
 :domain TimeFlies-domain
 :input (timeflies timeflies-machine data loaded in-form present)
 :output (loaded)
 :given (and (in-form tab data)
     (lessp (year-precision data) 100)
     (present timeflies-machine data))
 :makes (loaded timeflies data)
 :procedure do-import-tab)
```