3. Model Level Interface

We now describe the Cedar interface to the implementation of the Cypress data model. We assume that the reader is familiar with the basic conceptual data model, i.e., has read the previous section. Our presentation is therefore slightly different in this section: we describe the procedures in the database interface in roughly the order that a client will want to use them in a program. We present types and initialization, schema definition, the basic operations, and then queries.

It should be emphasized that the interface we are about to describe is only one possible implementation of the abstract data model described in Section 2. For example, we have chosen to implement a procedural interface called by Cedar programs, and to do type checking at run-time.

3.1 Types

In this subsection we describe the most important types in the interface. Less pervasive types are treated at the point where they are first used.

```
Entity: TYPE;
Relship: TYPE;
```

An Entity or Relship is not the actual database entity or relationship; they are *handles* for the actual database objects. All accesses to database objects are performed by calling interface procedures with the handles as parameters. Even comparisons of two entities for equality must be done in this way. The Entity and Relship handles are allocated from storage and automatically freed by the garbage collector when no longer needed.

```
Value: TYPE = REF ANY;
ValueType: TYPE;
Datatype: TYPE;
StringType, IntType, BoolType, AnyDomainType: DataType;
```

Storing Cedar data values in tuples presents several problems. First, since we would like to define a single operation to store a new value into a specified attribute of a Relship (for instance), there must be a single type for all values that pass through this "store-value" procedure. This is the type Value above, represented as untyped REFs in Cedar. The DataTypes will be discussed in the next section. Entities, strings, integers, and booleans are the types of values the system currently recognizes and allows as attribute values. More precisely, these four types are Entity, ROPE, REF INT, and REF BOOL. In the case of an entity-valued attribute, an attribute's type may be AnyDomainType or a specific domain may be specified. The latter is highly preferred, as AnyDomainType is a loophole in the type mechanism and limits the kinds of operations that can be performed automatically by the

database system or associated tools. We currently provide no mechanism to store compound Cedar data structures such as arrays, lists, or records in a database; the database system's data structuring mechanisms should be used instead. Cypress query operations such as RelationSubset cannot be composed upon data that appears as uninterpreted bits in the database.

Note that a Value may be either an Entity or a Datum. Some operations accept any Value, e.g. SetF; others require an Entity, e.g. NameOf. Others may require an Entity from a particular client-defined domain, e.g. a Person. We might think of the hierarchy of built-in and client defined types and instances of values like this:

Value type hierarchy	**Database representative of type**
Value (REF ANY)	ValueType
Datum	DatumType
ROPE	StringType
INT	IntType
BOOL	BoolType
Entity	AnyDomainType
person Entity	Person domain
employee Entity	Employee domain
... other client-defined entities ...	... other client-defined domains ...

As Cedar doesn't have a good mechanism for defining type hierarchies or new types for client-defined domains, most Cypress operations simply take a REF ANY or an Entity as argument, performing further type checking at run-time.

3.2 Transactions and segments

In this section we describe the basic operations to start up a database application's interaction with Cypress. The client application's data is stored in one or more segments, accessed under transactions. The Cypress system currently runs on the same machine as the client program, however transactions are implemented by the underlying file system which may reside on another machine. Data in remote segments may therefore be concurrently accessed by other instances of Cypress on other client machines.

A transaction is a sequence of read and write commands. The system supports the property that the entire sequence of commands executes *atomically* with respect to all other data retrieval and updates, that is, the transaction executes as if no other transactions were in progress at the same time. Because there may in fact be other transactions accessing the same data at the same time, it is possible that two transactions may deadlock, in which case one of them must be aborted. So the

price paid for concurrent access is that programs be prepared to retry aborted transactions.

The database system provides the capability of accessing a database stored on the same machine as the database client, using the Pilot file system or on Alpine file servers. We currently permit only one transaction per segment per instance of the database software on a client machine. That is, data in remote segments may concurrently be updated by application programs under separate transactions, but on the same machine transactions are used simply to make application transactions on their respective segments independent. This transaction-per-segment scheme is a major simplification of the Cypress package. In addition, as we shall see presently, nearly all Cypress procedures can automatically infer the appropriate segment and transaction from the procedure arguments, avoiding the need to pass the transaction or segment for every database operation.

Calls to Initialize, DeclareSegment, and OpenTransaction start the database session. A transaction is either passed in by the client, or created by the database package (the latter is just a convenience feature). The operation MarkTransaction below forms the end of a database transaction and the start of a new one. The operation AbortTransaction may be used to abort a transaction. Data in a database segment may not be read or updated until the segment and transaction have been opened. Clients must decide when to tell the system that a transaction is complete (with CloseTransaction), and must be prepared to deal with unsolicited notification that the current transaction has been aborted because of system failure or lock conflict.

The client's interaction with the database system begins with a call to Initialize:

```
Initialize: PROC[
 nCachePages: CARDINAL_ 256,
 nFreeTuples: CARDINAL_ 32,
 cacheFileName: ROPE_ NIL ];
```

Initialize initializes the database system and sets various system parameters: nCachePages tells the system how many pages of database to keep in virtual memory on the client's machine, nFreeTuples specifies the size to use for the internal free list of Entity and Relship handles, and cacheFileName is the name of the disk file used for the cache backing store. Any or all of these may be omitted in the call; they will be given default values. Initialize should be called before any other operation; the schema declaration operations generate the error DatabaseNotInitialized if this is violated.

Before database operations may be invoked, the client must open the segment(s) in which the data are stored. The location of the segment is specified by using the full path name of the file, e.g. "[MachineName]<Directory>SubDirectory>SegmentName.segment". Each segment has a unique name, the name of a Cedar ATOM which is used to refer to it in Cypress operation. The name of the Cedar ATOM is normally, though not necessarily, the same as that of the file in which it is

stored, except the extension ".segment" and the prefix specifying the location of the file is omitted in the ATOM. If the file is on the local file system, its name is preceded by "[Local]". For example, "[Local]Foo" refers to a segment file on the local disk named Foo.database; "[Alpine]<CedarDB>Baz" refers to a segment named Baz.segment on the <CedarDB> directory on the Alpine server. It is generally a bad idea to access database segments other than through the database interface. However, because segments are physically independent and contain no references to other files by file identifier or explicit addresses within files, the segment files may be moved from machine to machine or renamed without effect on their contents. If a segment file in a set of segments comprising a client database is deleted, the others may still be opened to produce a database missing only that segment's entities and relationships. A segment is defined by the operation DeclareSegment:

```
DeclareSegment: PROC[
  filePath: ROPE, segment: Segment, number: INT_ 0,
  readOnly: BOOL_ FALSE, version: Version_ OldOnly,
  nBytesInitial, nBytesPerExtent: LONG CARDINAL_ 32768]
  RETURNS [Segment];

Segment: TYPE = ATOM;
Version: TYPE = {NewOnly, OldOnly, NewOrOld};
```

The version parameter to DeclareSegment defaults to OldOnly to open an existing file. The signal IllegalFileName is generated if the directory or machine name is missing from fileName, and FileNotFound is generated at the time a transaction is opened on the segment if the file does not exist. If version NewOnly is passed, a new segment file will be created, erasing any existing one. In this case, a number assigned to the segment by the database administrator must also be passed. This number hack is necessitated by our current implementation of segments (it specifies the section of the database address space in which to map this segment). Please bear with us. Finally, the client program can pass version=NewOrOld to open a new or existing segment file; in this case the segment number must also be passed, of course.

The other parameters to DeclareSegment specify properties of the segment. If readOnly=TRUE, then writes are not permitted on the segment; any attempt to invoke a procedure which modifies data will generate the error ProtectionViolation. nBytesInitial is the initial size to assign to the segment, and nBytesPerExtent is the incremental increase in segment size used when more space is required for data in the file.

For convenience, a call is available to return the list of segments that have been declared in the current Cypress session:

```
GetSegments: PROC RETURNS[LIST OF Segment ];
```

A transaction is associated with a segment by using OpenTransaction:

```
OpenTransaction: PROC[
  segment: Segment,
  userName, password: ROPE_ NIL,
  useTrans: Transaction_ NIL ];
```

If useTrans is NIL then OpenTransaction establishes a new connection and transaction with the corresponding (local or remote) file system. Otherwise it uses the supplied transaction. The same transaction may be associated with more than one segment by calling OpenTransaction with the same useTrans argument for each. The given user name and password, or by default the logged in user, will be used if a new connection must be established.

Any database operations upon data in a segment before a transaction is opened or after a transaction abort will invoke the Aborted signal. The client should catch this signal on a transaction abort, block any further database operations and wait for completion of any existing ones. Then the client may re-open the aborted transaction by calling OpenTransaction. When the remote transaction is successfully re-opened, the client's database operations may resume.

Note that operations on data in segments under different transactions are independent. Normally there will be one transaction (and one or more segments) per database application program. A client may find what transaction has been associated with a particular segment by calling

```
TransactionOf: PROC [segment: Segment] RETURNS [Transaction];
```

Transactions may be manipulated by the following procedures:

```
MarkTransaction: PROC[trans: Transaction];

AbortTransaction: PROC [trans: Transaction];

CloseTransaction: PROC [trans: Transaction];
```

MarkTransaction commits the current database transaction, and immediately starts a new one. User variables which reference database entities or relationships are still valid.

AbortTransaction aborts the current database transaction. The effect on the data in segments associated with the segment is as if the transactions had never been started, the state is as it was just after the OpenTransaction call or the most recent MarkTransaction call. Any attempts to use variables referencing data fetched under the transaction will invoke the NullifiedArgument error. A

call to OpenTransaction is necessary to do more database operations, and all user variables referencing database items created or retrieved under the corresponding transaction must be re-initialized (they may reference entities or relationships that no longer exist, and in any case they are marked invalid by the database system).

A simple client program using the database system might have the form, then:

```
Initialize[];
DeclareSegment["[Local]Test", $Test];
OpenTransaction[$Test];
...
... database operations, including zero or more MarkTransaction calls ...
...
CloseTransaction[TransactionOf[$Test]];
```

3.3 Data schema definition

The definition of the client's data schema is done through calls to procedures defined in this section. The data schema is represented in a database as entities and relationships, and although *updates* to the schema must go through these procedures to check for illegal or inconsistent definitions, the schema can be *read* via the normal data operations described in the next section. Each domain, relation, etc., has an entity representative that is used in data operations which refer to that schema item. For example, we pass the domain entity when creating a new entity in the domain. The types of schema items are:

```
Domain, Relation, Attribute, Datatype, Index, IndexFactor: TYPE = Entity;
```

Of course, since the schema items are entities, they must also belong to domains; there are pre-defined domains, which we call *system domains*, in the interface for each type of schema entity:

```
DomainDomain, RelationDomain, AttributeDomain, DatatypeDomain, IndexDomain: Domain;
```

There are also pre-defined system relations, which contain information about sub-domains, attributes, and indices. Since these are not required by the typical (application-specific) database client, we defer the description of the system relations to Section 3.6.

In general, any of the data schema may be extended or changed at any time; i.e., data operations and data schema definition may be intermixed. However, there are a few specific ordering constraints on schema definition we will note shortly. Also, the database system optimizes for better performance if the entire schema is defined before any data are entered. The interactive schema editing

tool described in the database tools documentation allows the schema to be changed regardless of ordering constraints and existing data, by recreating schema items and copying data invisibly to the user when necessary.

All the data schema definition operations take a Version parameter which specifies whether the schema element is a new or existing one. The version defaults to allowing either (NewOrOld): i.e., the existing entity is returned if it exists, otherwise it is created. This feature avoids separate application code for creating the database schema the first time the application program is run.

```
DeclareDomain: PROC [name: ROPE, segment: Segment,
  version: Version_ NewOrOld, estRelations: INT_ 5] RETURNS [d: Domain];

DeclareSubType: PROC[sub, super: Domain];
```

DeclareDomain defines a domain with the given name in the given segment and returns its representative entity. If the domain already exists and version=NewOnly, the signal AlreadyExists is generated. If the domain does not already exist and version=OldOnly, then NIL is returned. The parameter estRelations is used to estimate the largest number of relations in which entities of this domain are expected to participate.

The client may define one domain to be a subtype of another by calling DeclareSubType. This permits entities of the subdomain to participate in any relations in which entities of the superdomains may participate. All client DeclareSubType calls should be done before declaring relations on the superdomains (to allow some optimizations). The error MismatchedSegment is generated if the sub-domain and super-domain are not in the same segment.

```
DeclareRelation: PROC [
  name: ROPE, segment: Segment, version: Version_ NewOrOld] RETURNS [r: Relation];

DeclareAttribute: PROC [
  r: Relation, name: ROPE, type: ValueType_ NIL,
  uniqueness: Uniqueness _ None,  length: INT_ 0,
  link: {Linked, Unlinked, Colocated, Remote}_ yes, version: Version_ NewOrOld]
  RETURNS[a: Attribute];

Uniqueness: TYPE = {NonKey, Key, KeyPart, OptionalKey};
```

DeclareRelation defines a new or existing relation with the given name in the given segment and returns its representative entity. If the relation already exists and version=NewOnly, the signal AlreadyExists is generated. If the relation does not already exist and version=OldOnly, then NIL is returned.

DeclareAttribute is called once for each attribute of the relation, to define their names, types, and uniqueness. If version=NewOrOld and the attribute already exists, Cypress checks that the new type, uniqueness, etc. match the existing attribute. The error MismatchedExistingAttribute is generated if there is a discrepancy. The attribute name need only be unique in the context of its relation, not over all attributes. Note this is the only exception to the data model's rule that names be unique in a domain. Also note that we could dispense with DeclareAttribute altogether by passing a list into the DeclareRelation operation; we define a separate procedure for programming convenience.

The attribute type should be a ValueType, i.e. it may be one of the pre-defined types (IntType, StringType, BoolType, AnyDomainType) or the entity representative for a domain. For pre-defined types, the actual values assigned to attributes of the relationship instances of the relation must have the corresponding type: `REF INT, ROPE, REF BOOL`, or `Entity`. If the attribute has a domain as type, the attribute values in relationships must be entities of that domain or some sub-domain thereof. The type is permitted to be one of the pre-defined system domains such as the DomainDomain, thereby allowing client-defined extensions to the data schema (for example, a comment for each domain describing its purpose).

The attribute uniqueness indicates whether the attribute is a key of the relation. If its uniqueness is NonKey, then the attribute is not a key of the relation. If its uniqueness is OptionalKey, then the system will ensure that no two relationships in r have the same value for this attribute (if a value has been assigned). The error NonUniqueKeyValue is generated if a non-unique key value results from a call to the SetP, SetF, SetFS, or CreateRelship procedures we define later. Key acts the same as OptionalKey, except that in addition to requiring that no two relationships in r have the same value for the attribute, it requires that every entity in the domain referenced by this attribute must be referenced by a relationship in the relation: the relationships in the relation and the entities in the domain are in one-to-one correspondence. Finally, if an attribute's uniqueness is KeyPart, then the system will ensure that no two relationships in r have the same value for *all* key attributes of r, though two may have the same values for some subset of them.

The length and link arguments to DeclareAttribute have no functional effect on the attribute, but are hints to the database system implementation. For StringType fields, length characters will be allocated for the string within the space allocated for a relationship in the database. There is no upper limit on the size of a string-valued attribute; if it is longer than length, it will be stored separately from the relationship with no visible effect except for the performance of database applications. The link field is used only for entity-valued fields; it suggests whether the database system should link together relationships which reference an entity in this attribute. In addition, it can suggest that the relationships referencing an entity in this attribute be physically co-located as well as linked. Again, its logical effect is only upon performance, not upon the legal operations.

```
DestroyRelation: PROC[r: Relation];

DestroyDomain: PROC[d: Domain];

DestroySubType: PROC[sub, super: Domain];
```

Relations, domains, and subdomain relationships may be destroyed by calls to the above procedures. Destroying a relation destroys all of it relationships. Destroying a domain destroys all of its entities and also any relationships which reference those entities. Destroying a sub-domain relationship has no effect on existing domains or their entities; it simply makes entities of domain sub no longer eligible to participate in the relations in which entities of domain super can participate. Existing relationships violating the new type structure are allowed to remain. Existing relations and domains may only be modified by destroying them with the procedures above, with one exception: the operation ChangeName (described in Section 3.4) may be used to change the name of a relation or domain.

```
DeclareIndex: PROC [
  relation: Relation, indexedAttributes: AttributeList, version: Version];
```

DeclareIndex has no logical effect on the database; it is a performance hint, telling the database system to create a B-Tree index on the given relation for the given indexedAttributes. The index will be used to process queries more efficiently. Each index key consists of the concatenated values of the indexedAttributes in the relationship the index key references. For entity-valued attributes, the value used in the key is the string name of the entity. The version parameter may be used as in other schema definition procedures, to indicate a new or existing index. If any of the attributes are not attributes of the given relation then the signal IllegalIndex is generated.

The optimal use of indices, links, and colocation, as defined by DeclareIndex and DeclareAttribute, is complex. It may be necessary to do some space and time analysis of a database application to choose the best trade-off, and a better trade-off may later be found as a result of unanticipated access patterns. Note, however, that a database may be rebuilt with different links, colocation, or indices, and thanks to the data independence our interface provides, existing programs will continue to work without change.

If a relation is expected to be very small (less than 100 relationships), then it might reasonably be defined with neither links nor indices on its attributes. In the typical case of a larger relation, one should examine the typical access paths: links are most appropriate if relationships that pertain to particular entities are involved, indices are more useful if sorting or range queries are desired.

B-tree indices are always maintained for domains; that is, an index contains entries for all of the entities in a domain, keyed by their name, so that sorting or lookup by entity name is quick. String comparisons are performed in the usual lexicographic fashion.

```
DeclareProperty: PROC [
  relationName: ROPE, of: Domain, type: ValueType,
  uniqueness: Uniqueness_ None, version: Version_ NewOrOld]
  RETURNS [property: Attribute];
```

DeclareProperty provides a shorthand for definition of a binary relation between entities of the domain "of" and values of the specified type. The definitions of type and uniqueness are the same as for DeclareAttribute. A new relation relationName is created, and its attributes are given the names "of" and "is". The "is" attribute is returned, so that it can be used to represent the property in GetP and SetP defined in the next section.

3.4 Basic operations on entities and relationships

In this section, we describe the basic operations on entities and relationships; we defer the operations on domains and relations to the next section.

A number of error conditions are common to all of the procedures in this section. Since values are represented as REF ANYs, all type checking must currently be done at run-time. The procedures in this section indicate illegal arguments by generating the errors IllegalAttribute, IllegalDomain, IllegalRelation, IllegalValue, IllegalEntity, and IllegalRelship, according to the type of argument expected. The error NILArgument is generated if NIL is passed to any procedure that cannot accept NIL for that argument. The error NullifiedArgument is generated if an entity or relationship is passed in after it has been deleted or rendered invalid by transaction abort or close.

```
DeclareEntity:  PROC[
  d: Domain, name: ROPE_ NIL, version: Version_ NewOrOld]
  RETURNS [e: Entity];
```

DeclareEntity finds or creates an entity in domain d with the given name. The name may be omitted if desired, in which case an entity with a unique name is automatically created. If version is OldOnly and an entity with the given name does not exist, NIL is returned. If version is NewOnly and an entity with the given name already exists, the signal NonUniqueEntityName is generated.

```
DeclareRelship: PROC [
  r: Relation, avl: AttributeValueList_ NIL, version: Version_ NewOrOld]
  RETURNS [Relship];
```

DeclareRelship finds or creates a relship in r with the given attribute values. If version is NewOnly, a new relship with the given attribute values is generated. If version is OldOnly, the relship in r with the given attribute values is returned if it exists, otherwise NIL is returned. If

version is NewOrOld, the relship with the given attribute values is returned if it exists, otherwise one is created. If the creation of a new relship violates the key constraints specified by DeclareAttribute, the signal NonUniqueAttributeValue is generated.

```
DestroyEntity: PROC[e: Entity];
```

DestroyEntity removes e from its domain, destroys all relationships referencing it, and destroys the entity representative itself. Any client variables that reference the entity automatically take on the null value (Null[e] returns TRUE), and cause error NullifiedArgument if passed to database system procedures. After an entity is destroyed, its old name may be re-used in creating a new one.

```
DestroyRelship: PROC[t: Relship];
```

DestroyRelship removes t from its relation, and destroys it. Any client variables that reference the relationship automatically take on the null value, and will cause error NullifiedArgument if subsequently passed to database system procedures.

```
SetF: PROC[t: Relship, a: Attribute, v: Value];
```

SetF assigns the value v to attribute a of relationship t. If the value is not of the same type as the attribute (or a subtype thereof if the attribute is entity-valued), then the error MismatchedAttributeValueType is generated. If a is not an attribute of t's relation, IllegalAttribute is generated.

```
GetF: PROC[t: Relship, a: Attribute] RETURNS [Value];
```

GetF retrieves the value of attribute a of relationship t. If a is not an attribute of t's relation, error IllegalAttribute is generated. The client should use the V2x routines described in the next section to coerce the value into the expected type.

```
SetFS: PROC [t: Relship, a: Attribute, v: ROPE];
GetFS: PROC[t: Relship, a: Attribute] RETURNS [ROPE];
```

GetFS and SetFS provide a convenient veneer on top of GetF and SetF that provide the illusion that all relation attributes are string-valued. The effect is something like the Relational data model, and is useful for applications such as a relation displayer and editor that deal only with strings. The semantics of GetFS and SetFS depend upon the actual type of the value v of attribute a:

type	GetFS returns	SetFS assigns attribute to be
StringType	the string v	the string v
IntType	v converted to decimal string	the string converted to decimal integer
BoolType	"TRUE" or "FALSE"	true if "TRUE", false if "FALSE"
a domain D	the name of the ref'd entity (or null string if v is NIL)	the entity with name v (or NIL if v is null string)
AnyDomainType	same, but includes domain: <domain-name>:<entity-name>	the entity with the given domain and name (or NIL if v is null string)

The same signals generated by GetF and SetF, such as IllegalAttribute, can also be generated by these procedures. The string NIL represents the undefined value. The signal NotFound is generated in the last case above if no entity with the given name is found.

```
NameOf: PROC [e: Entity] RETURNS [s: ROPE];
ChangeName: PROC [e: Entity, s: ROPE];
```

NameOf and ChangeName retrieve or change the name of an entity, respectively. They generate the signal IllegalEntity if e is not an entity.

ChangeName should be used with caution. It is not quite equivalent to destroying and re-creating an entity with the new name but the same existing relationships referencing it. ChangeName is considerably faster than that, and furthermore entity-valued variables which reference the entity are *not* nullified by ChangeName, though they would be by DestroyEntity. These features should be a help, not a hindrance. However, changing an entity name may invalidate references to the entity from outside the segment, e.g. in another segment or in some application-maintained file such as a log of updates.

```
DomainOf: PROC[e: Entity] RETURNS [Domain];
RelationOf: PROC[t: Relship] RETURNS [Relation];
```

DomainOf and RelationOf can be used to find the entity representative of an entity's domain or a relationship's relation, respectively. The signal IllegalEntity is generated if e is not an entity. The signal IllegalRelship is generated if r is not a relationship.

```
SegmentOf: PROC[e: Entity] RETURNS [Segment];
```

SegmentOf returns the segment in which an entity is stored. It can be applied to domain, relation, or attribute entities.

```
Eq: PROC [e1: Entity, e2: Entity] RETURNS [BOOL];
```

Eq returns TRUE iff the same database entity is referenced by e1 and e2. This is *not* equivalent to the Cedar expression "e1 = e2", which computes Cedar REF equality. If e1 and e2 are in

different segments, Eq returns true iff they have the same name and their domains have the same name.

```
Null: PROC [x: EntityOrRelship] RETURNS [BOOL];
```

Null returns TRUE iff its argument has been destroyed, is NIL, or has been invalidated by abortion of the transaction under which it was created.

```
GetP: PROC [e: Entity, als: Attribute, aOf: Attribute_ NIL] RETURNS [Value];
SetP: PROC [e: Entity, als: Attribute, v: Value, aOf: Attribute_ NIL] RETURNS[Relship];
```

GetP and SetP are convenience routines for a common use of relationships, to represent "properties" of entities. Properties allow the client to think of values stored in relationships referencing an entity as if they are directly accessible fields (or "properties") of the entity itself. See the figure on page 15 illustrating properties. GetP finds a relationship whose from attribute is equal to e, and returns that relationship's to attribute. The from attribute may be defaulted if the relation is binary, it is assumed to be the other attribute of to's relation. If it is not binary, the current implementation will find the "first" other attribute, where "first" is defined by the order of the original calls calls to DeclareAttribute. SetP defaults the from attribute similarly to GetP, but operates differently depending on whether from is a key of the relation. Whether it is a key or not, any previous relationship that referenced e in the from attribute is automatically deleted. In either case, a new relationship is created whose from attribute is e and whose to attribute is v. SetP returns the relationship it creates for the convenience of the client. GetP and SetP can generate the same errors as SetF and GetF, e.g. if e is not an entity or to is not an attribute. In addition, GetP and SetP can generate the error IllegalProperty if to and from are from different relations. GetP generates the error MismatchedPropertyCardinality if more than one relationship references e in the from attribute; if *no* such relationships exist, it returns a null value of the type of the to attribute. SetP allows any number of existing relationships referencing e; it simply adds another one (when from is a key, of course, there will always be one relationship).

```
GetPList: PROC [e: Entity, to: Attribute, from: Attribute_ NIL] RETURNS [LIST OF Value];
SetPList: PROC [e: Entity, to: Attribute, vl: LIST OF Value, from: Attribute_ NIL];
```

GetPList and SetPList are similar to GetP and SetP, but they assume that any number of relationships may reference the entity e with their from attribute. They generate the signal MismatchedPropertyCardinality if this is not true, i.e. the from attribute is a key. GetPList returns the list of values of the to attributes of the relationships that reference e with their from attribute. Cedar has LISP-like list manipulation facilities. SetPList destroys any existing relationships that reference e with their from attribute, and creates Length[vl] new ones, whose from attributes reference e and

whose to attributes are the elements of vl. GetPList and SetPList may generate any of the errors that GetF and SetF may generate, and the error IllegalProperty if to and from are from different relations.

Note that the semantics of SetPList are not quite consistent with the semantics of SetP. SetPList *replaces* the current values associated with a "property" with the new values (i.e., destroys and re-creates relationships); SetP *adds* a new property value, unless the aOf attribute is a key, in which case it replaces the current value. The semantics are defined in this way because this has proven the most convenient in our application programs.

Examples of the use of the property procedures for data access can be found in Section 4.3. Properties are also useful for obtaining information about the data schema. For example, GetP[a, aRelationIs] will return the attribute a's relation, and GetPList[d, aTypeOf] will return all the attributes that can reference domain d.

```
E2V: PROC[e: Entity] RETURNS[v: Value];
B2V: PROC[b: BOOLEAN] RETURNS[v: Value];
I2V: PROC[i: LONG INTEGER] RETURNS[v: Value];
S2V: PROC[s: ROPE] RETURNS[v: Value];
```

The x2V routines convert the various Cedar types to Values. The conversion is not normally required for ropes and entities since the compiler will widen these into the REF ANY type Value.

```
V2E: PROC[v: Value] RETURNS[Entity];
V2B: PROC[v: Value] RETURNS[BOOLEAN];
V2I: PROC [v: Value] RETURNS[LONG INTEGER];
V2S: PROC [v: Value] RETURNS[ROPE];
```

The V2x routines convert Values to the various Cedar types. The MismatchedValueType error is raised if the value is of the wrong type. It is recommended that these routines be used rather than user-written NARROWs, as the representation of Values may change. Also, NARROWs of opaque types don't yet work in the Cedar compiler.

3.5 Query operations on domains and relations

In this section we describe queries upon domains and relations: operations that enumerate entities or relationships satisfying some constraint.

```
RelationSubset: PROC[
  r: Relation, constraint: AttributeValueList_ NIL]
  RETURNS [RelshipSet];

NextRelship: PROC[rs: RelshipSet] RETURNS [Relship];

PrevRelship: PROC[rs: RelshipSet] RETURNS [Relship];

ReleaseRelshipSet: PROC [rs: RelshipSet];

AttributeValueList: TYPE = LIST OF AttributeValue;
AttributeValue: TYPE = RECORD [
   attribute: Attribute,
   low: Value,
   high: Value_ NIL -- omitted where same as low or not applicable --];
```

The basic query operation is RelationSubset. It returns a generator of all the relationships in relation r which satisfy a constraint list of attribute values. The relationships are enumerated by calling NextRelship repeatedly; it returns NIL when there are no more relationships. PrevRelship may similarly be called repeatedly to back the enumeration up, returning the previous relationship; it returns NIL if the enumeration is at the beginning. ReleaseRelshipSet should be called when the client is finished with the query.

The constraint list may be NIL, in which case all of the relationships in r will be enumerated. Otherwise, relationships which satisfy the concatenation of constraints on attributes in the list will be enumerated. If an index exists on some subset of the attributes, the relationships will be enumerated sorted on the concatenated values of those attributes. For a StringType, IntType, or TimeType attribute a of r, the contraint list may contain a record of the form [a, b, c] where the attribute value must be greater or equal to b and less than or equal to c to satisfy the constraint. For any type of attribute, the list may contain a record of the form [a, b] where the value of the attribute must exactly equal b. The Cedar ROPE literals "" and "\377" may be used in queries as an infinitely large and infinitely small string, respectively. The signal MismatchedAttributeValueType is generated by RelationSubset if one of the low or high values in the list is of a different type than its corresponding attribute.

```
DomainSubset: PROC[
  d: Domain,
  lowName, highName: ROPE_ NIL,
  searchSubDomains: BOOL_ TRUE,
  searchSegment: Segment_ NIL]
```

```
    RETURNS [EntitySet];

  NextEntity: PROC[EntitySet] RETURNS [Entity];

  PrevEntity: PROC[EntitySet] RETURNS [Entity];

  ReleaseEntitySet: PROC[EntitySet];
```

DomainSubset enumerates all the entities in a domain. If lowName and highName are NIL, the entire domain is enumerated, in no particular order. Otherwise, only those entities whose names are lexicographically greater than lowName and less than highName are enumerated, in lexicographic order. If searchSubDomains is TRUE, subdomains of d are also enumerated. Each subdomain is sorted separately. The searchSegment argument is currently only used if d is one of the system domains, e.g. the Domain domain. It is used to specify which segment to search.

Analogously to relation enumeration, NextEntity and PrevEntity may be used to enumerate the entities returned by DomainSubset, and ReleaseEntitySet should be called upon completion.

```
  GetDomainRefAttributes: PROC [d: Domain] RETURNS [AttributeList];
```

This procedure returns a list of all attributes, of any relation defined in d's segment, which reference domain d or one of its superdomains. The list does not include AnyDomainType attributes, which can reference any domain. GetDomainRefAttributes is implemented via queries on the data schema. GetDomainRefAttributes is useful for application-independent tools; most specific applications can code-in the relevant attributes.

```
  GetEntityRefAttributes: PROC [e: Entity] RETURNS [AttributeList];
```

This procedure returns a list of all attributes in which some existing relationship actually references e, including AnyDomainType attributes.

3.6 System domains and relations

In this section we describe what one might call the *schema schema*, the pre-defined system domains and relations which constitute the data schema for client-defined domains and relations. The typical database application writer may skip this section, since the schema declaration operations defined in Section 3.3 are adequate when the data schema is completely defined and known at the time a program is written. The system domains and relations we describe in this section are most useful for general-purpose tools (e.g. for displaying, querying, or dumping *any* database), where the tools must

examine the data schema "on the fly".

As noted earlier, the permanent repository for data describing user-defined data in a database is the database's data schema, represented by schema entities and relationships. Schema entities are members of one of the pre-defined system domains: DomainDomain, RelationDomain, DatatypeDomain, and so on. Every client-defined domain, relation, or attribute contains a representative entity in these domains. Client-defined datatypes are not currently permitted, so the only entities in the DataType domain are the pre-defined IntType, StringType, and BoolType.

The information about the client-defined domains and attributes are encoded by relationships in the database. Domains participate in the system relation dSubType, which encodes a domain type hierarchy:

```
dSubType: Relation;
  dSubTypeOf: Attribute; -- the domain in this attribute is a super-type of
  dSubTypeIs: Attribute; -- the domain in this attribute
```

The dSubType has one element per direct domain-subdomain relationship, it does not contain the transitive closure of that relation. However, it is guaranteed to contain no cycles. That is, the database system checks that there is no set of domains d_1, d_2, ... d_N, N>1, such that d_1 is a subtype of d_2, d_2 is a subtype of d_3, and so on to d_N, and d_1=d_N. The dSubType may define a lattice as opposed to a tree, i.e. the sSubType attribute is not a key of the relation.

The information about attributes is encoded as binary relations, one relation for each argument to the DeclareAttribute procedure defining properties of the attribute. The names are easy to remember; for each argument, e.g. Foo, we define the aFoo relation, with attributes aFooOf and aFooIs. The aFooIs attribute is the value of that argument, and the aFooOf attribute is [the entity representative of] the attribute it pertains to. Thus we have the following relations:

```
aRelation: PUBLIC READONLY Relation; -- Specifies attribute - relation correspondence:
  -- [aRelationOf: KEY Attribute, aRelationIs: Relation]
  aRelationOf: PUBLIC READONLY Attribute; -- attribute whose relation we are specifying
  aRelationIs: PUBLIC READONLY Attribute; -- the relation of that attribute

aType: PUBLIC READONLY Relation; -- Specifies types of relation attributes:
  -- [aTypeOf: KEY Attribute, aTypeIs: ValueType]
  aTypeOf: PUBLIC READONLY Attribute; -- the attribute
  aTypeIs: PUBLIC READONLY Attribute; -- domain or datatype of the attribute

aUniqueness: PUBLIC READONLY Relation; -- Specifies attribute value uniqueness:
 -- [aUniquenessOf: KEY Attribute, aUniquenessIs: INT LOOPHOLE[Uniqueness]]
```

```
    aUniquenessOf: PUBLIC READONLY Attribute; -- the attribute
    aUniquenessIs: PUBLIC READONLY Attribute; -- INT for Uniqueness: 0=None, 1=Key, etc.

  aLength: PUBLIC READONLY Relation; -- Specifies length of attributes:
    -- [aLengthOf: KEY Attribute, aLengthIs: INT]
    aLengthOf: PUBLIC READONLY Attribute; -- the attribute
    aLengthIs: PUBLIC READONLY Attribute;  -- INT corresponding to attribute's length

  aLink: PUBLIC READONLY Relation; -- Specifies whether attribute is linked:
    -- [aLinkOf: KEY Attribute, aLinkIs: INT]
    aLinkOf: PUBLIC READONLY Attribute; -- the attribute
    aLinkIs: PUBLIC READONLY Attribute;  -- 0=unlinked, 1=linked, 2 =colocated
```

The final set of system relations pertain to *index factors*. Each index on a relation is defined to include one or more attributes of a relation. For each attribute in the index, there is an index factor entity. For each index, there is an index entity. Each index factor is associated with exactly one index and exactly one attribute. Indices may have many index factors, however, and an attribute may be associated with more than one index factor, since attributes may participate in multiple indices. The two relations pertaining to indices map indices on to their index factors, and index factors to the attributes they index:

```
  ifIndex: PUBLIC READONLY Relation; -- Specifies the index factors for each index
    -- [ifIndexOf: KEY IndexFactor, ifIndexIs: Index]
    ifIndexOf: PUBLIC READONLY Attribute;  -- the index factor
    ifIndexIs: PUBLIC READONLY Attribute;  -- index of the factor

  ifAttribute: PUBLIC READONLY Relation; -- Specifies attribute index factor corresponds to
    -- [ifAttributeOf: KEY IndexFactor, ifAttributeIs: Attribute]
    ifAttributeOf: PUBLIC READONLY Attribute; -- the index factor
    ifAttributeIs: PUBLIC READONLY Attribute; -- the attribute this factor represents
```

The relations on attributes, index factors, and domains can be queried with the RelationSubset or GetPList operations. For example, GetP[a, aRelationIs] returns the attribute a's relation. GetPList[r, aRelationOf] returns the relation r's attributes. RelationSubset[dSubType, LIST[[dSubTypeIs, d]]] will enumerate all the dSubType relationships in which d is the subtype.

As noted earlier, the data schema (attributes, relations, domains, indices, index factors, and relations pertaining to these) may only be read, not written by the database client. In order to ensure the consistency of the schema, it must be written indirectly through the schema definition procedures: DeclareDomain, DeclareRelation, DeclareAttribute, and DeclareSubType. Attempts to perform

updates through operations such as SetP result in the error ImplicitSchemaUpdate.

3.7 Errors

When a database system operation invokes an error, the SIGNAL Error is generated, with an error code indicating the type of error that occured. The error code is a Cedar enumerated type:

```
Error: SIGNAL [code: ErrorCode];
ErrorCode: TYPE = {
        AlreadyExists, -- Entity already exists and client said version=NewOnly
        BadUserPassword, -- On an OpenTransaction
        DatabaseNotInitialized, -- Attempt to do operation without calling Initialize
        FileNotFound, -- No existing segment found with given name
        IllegalAttribute, -- Attribute not of the given relship's Relation or not an attribute
        IllegalValueType, -- Type passed DeclareAttribute is not datatype or domain
        IllegalDomain, -- Argument is not actually a domain
        IllegalFileName, -- No directory or machine given for segment
        IllegalEntity, -- Argument to GetP, or etc., is not an Entity
        IllegalRelship, -- Argument to GetF, or etc., is not a Relship
        IllegalRelation, -- Argument is not a relation
        IllegalSegment, -- Segment passed to DeclareDomain, or etc., not yet declared
        IllegalString, -- Nulls not allowed in ROPEs passed to the database system
        IllegalSuperType, -- Can't define subtype of domain that already has entities
        IllegalValue, -- Value is not REF INT, ROPE, REF BOOL, or Entity
        IllegalValueType, -- Type passed DeclareAttribute is not datatype or domain
        ImplicitSchemaUpdate, -- Attempt to modify schema with SetP, DeclareEntity, etc.
        InternalError, -- Impossible internal state (possibly bug or bad database)
        MismatchedProperty, -- aOf and aIs attribute not from the same relation
        MismatchedAttributeValueType, -- Value not same type as required (SetF)
        MismatchedExistingAttribute, -- Existing attribute is different (DeclareAttribute)
        MismatchedExistingSegment, -- Existing segment is different (DeclareSegment)
        MismatchedPropertyCardinality, -- Did GetP with aOf that is not a Key
        MismatchedSegment, -- Attempt to create ref across segment boundary (SetF)
        MismatchedValueType, -- value passed V2E, V2I, etc. not of expected type
        MultipleMatch, -- More than one relationship satisfied avl on DeclareRelship.
        NonUniqueEntityName, -- Entity in domain with that name already exists
        NonUniqueKeyValue, -- Relship already exists with that value
        NotFound, -- Version is OldOnly but no such Entity, Relation, or etc found
        NotImplemented, -- Action requested is not yet implemented
        NILArgument, -- Attempt to perform operation on NIL argument
        NullifiedArgument, -- Entity or relationship has been deleted or invalidated
        ProtectionViolation, -- Read or write to segment not permitted this user.
        SegmentNotDeclared, -- Attempt to open transaction w/o DeclareSegment
        ServerNotFound -- File server does not exist or does not respond
        };
```

In this report, the expression "generates the error X" means that the SIGNAL Error is generated with code=X. Unless otherwise specified, the client may CONTINUE from the signal, aborting the operation in question. Signals should not be RESUMEd except by a wizard who knows the result of proceeding with an illegal operation.

Two special signals are associated with the file system level:

```
Aborted: SIGNAL [trans: Transaction];
Failure: SIGNAL [why: ATOM, server: ROPE];
```

The Aborted signal can be generated by any database operation, and indicates that the transaction has been aborted. The client must call AbortTransaction and may then call OpenTransaction to proceed. The Failure signal is generated when a transaction cannot be open due to server failure or communication difficulties.