Comparing Specification Paradigms: Gypsy and Z

William D. Young

Technical Report 45 June, 1989

This paper will be presented at the 12th National Computer Security Conference, Baltimore, Maryland, October 10-13, 1989.

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

1. Introduction

The application of formal methods to the analysis of computing systems promises to provide higher and higher levels of assurance as the sophistication of our tools and techniques increases. But evolution of the state of the art of formal program analysis is matched by increasing demands upon the technology. In the security arena advances in program verification methodologies, automated reasoning systems, specification techniques, and security modeling have been met with continuing reassessment of acceptable levels of assurance. System developers contemplating certification at the A1 level as outlined in the *Trusted Computer Systems Evaluation Criteria* [3], for example, can expect that the assurance requirements will become more rigorous with each year that passes.

Conversely, the desire for enhanced assurance drives the evolution of tools and techniques for providing it. One way to assure that technology keeps pace with evolving expectations is by continually re-evaluating our entrenched tools and techniques in relation to possible alternatives. The alternation of evaluation with informed refinement and selection can incrementally improve the research environment for all.[1] The verification community has been quite willing to compare and contrast various technologies and systems [15, 2, 16, 13] though it is unclear how much these comparisons have led to specific changes.

One of the most entrenched tools for providing assurance in the security area is the Gypsy Verification Environment [7] (GVE). The GVE is one of two systems endorsed by the National Computer Security Center for use in meeting the verification requirements for A1 certification. It has been used extensively in secure system specification and verification projects including the Encrypted Packet Interface [21], Message Flow Modulator [8], Honeywell SCOMP [5], Honeywell LOCK [1], and ACCAT Guard [14].

The Z specification language [9, 22] evolved within the Programming Research Group at Oxford University. We are not aware of its use within the security community though it has been used to specify significant software systems including a subset of the Unix filing system [18], the Computer Aided Visitor Information and Retrieval System [4], the ICL Data Dictionary [23], and a CICS System at IBM in the U.K. These examples suggest that Z might provide a viable specification language for secure systems. One goal of our research was to investigate this suggestion.

We present a comparison of the Gypsy and Z specification languages in the context of a nontrivial

[1]This paradigm of scientific process is often blocked by prejudice for or against certain research directions, the personal and financial investments researchers and user communities have in those directions, and the momentum of ongoing system development projects.

example. Our example is a previous specification of a subset of the Unix file system functionality [18][2] in Z and the translation of this specification into Gypsy. We compare and contrast the two specifications. On the basis of this comparison, certain conclusions are drawn which we hope can suggest refinements to the two languages and possibly a direction for future language designs which will avoid the pitfalls and capitalize on the strong points of each.

2. The Two Languages

2.1 Gypsy

Gypsy [6][3] is a *program description language* composed of two strongly intersecting components: a programming language and a specification language. Some parts of the language are used for programming, some for specification, and some for both. Among other advantages, this provides a common framework for expressing specifications and programs and obviates the need for elaborate mappings from specifications to programs. A potential disadvantage is that it is quite easy in Gypsy to write specifications which are semantically quite similar to the implementation.

Gypsy is descended from Pascal [11] and contains features for data abstraction, condition handling, dynamic memory management, and concurrency. The specification component of the language contains the full expressive power of the predicate calculus and the ability to write recursive functions. Specifications may be written as Floyd-Hoare style program annotations, algebraic-style axioms, or state machine descriptions.

The Gypsy Verification Environment [7] is a collection of software tools which form a development environment for creating, specifying, maintaining, and verifying Gypsy programs. These tools include a parser, verification condition generator, interactive proof checker, and algebraic simplifier. Gypsy is fully described in [6] and a methodology for using the language effectively is documented in [7].

[2]We are experts in the use of Gypsy but did not feel confident to write a creditable Z specification. We chose a problem which had been specified by Z experts to present Z favorably; we wish to make it absolutely clear from the outset that all of the Z text in this paper is taken almost verbatim from [18]. It is included here only to make the current presentation self-contained.

[3]We devote somewhat less attention to the description of Gypsy than to Z since most security researchers have at least a passing acquaintance with Gypsy.

2.2 The Z Specification Language

Key to understanding Z is the designers' "conviction that real software *can* be specified and that ordinary mathematics is the proper tool" [9]. Z purports to offer a standard mathematical notation which is "easy for a scientifically trained reader to understand; is rigorous; denotes rich concepts; and is an open notation, because you can enlarge it at will" [9].

The basic structuring concept is the *schema* [24]. A schema is an association of variable declarations and *observations* about those variables. An observation merely expresses some relation among variables. An observation can be viewed as placing a constraint upon any implementation of the specification. Schemata can be written in either a tabular or linear form; the tabular form seems to be the preferred form.

Consider the following schema for a portion of a specification of a symbol table abstraction [9].

```
___LOOKUP_____________________________
|    st, st' : ST
|    s?  : SYM
|    v!  : VAL
|_________________
|
|    st' = st
|    s? ∈ dom (st)
|    v! = st (s?)
|_____________________________________
```

The top portion of the schema defines a collection of variables: a variable **st** which is a mapping from symbols (**SYM**) to values (**VAL**), an input variable **s?**, and an output variable **v!** of the indicated types. The intelligibility of the schema relies heavily upon certain conventions. The primed variable **st'** is assumed to be the final value of **st**; variables suffixed with "?" and "!" are assumed to be for input and output, respectively.

The bottom portion of the schema is a collection of *observations* stating relations among the variables. These are merely predicate calculus expressions involving the variables of the schema and may involve any of the standard operations of predicate calculus, elementary set theory, or mathematics. A standard notation is defined in [9]. In the example above, the observations indicate that the state variable is unchanged by the **LOOKUP** operation. A precondition of the operation is that input symbol **s?** is in the domain of the state mapping. The output value **v!** is the result of accessing the value currently associated with **s?** in the state. Notice that the specification is highly nonprocedural and places no constaints on the implementation other than the logical consequences of the observations.

The top portion may also include the names of other schemata indicating that these are to be included as subparts of the current schema. Common variables are shared and the collections of observations are conjoined. Inclusion is the simplest operation in a schema calculus which permits building up complex and well structured specifications by defining and combining schemata. The schema calculus provides a notation for expressing complex schemata compactly; the schema operations seem to be entirely eliminable in favor of a (possibly quite large) list of variable declarations and observations.

3. The Unix File System Example

A specification in Z of part of the functionality of the Unix filing system is given in [18]. The system modeled is UNIX Level 6. Operations covered include nine system calls—**`read`**, **`write`**, **`create`**, **`seek`**, **`open`**, **`close`**, **`fstat`**, **`link`**, and **`unlink`**—and the commands **`ls`** and **`move`**. The specification seems to have been intended as a tutorial example of the use of Z and proceeds by defining a series of progressively more elaborate mechanisms for accessing and manipulating files. At each level, additional complexity is added by defining new schemata from variable declarations, observations and previous schemata using logic and the schema calculus.

We developed a specification of the same functionality in Gypsy trying to follow as closely as possible the development style while still constructing a reasonable Gypsy specification. Our approach was to define a sequence of Gypsy scopes reflecting the added functionality at each step in the development of the specification. For example, the authors of the Z specification define an initial mechanism for reading and storing files considered as byte sequences. They then elaborate this into a mechanism for reading and storing files with a level of indirection representing the filing system. To mimic this structure, we first wrote a Gypsy scope modeling the reading and storing of files as byte sequences. A subsequent scope used the types and procedures defined there to define other procedures adding the level of indirection. Our goal was a Gypsy specification which would be as easy as possible to compare to the Z specification. In this section we illustrate the two specifications by considering the successive elaborations of **`WRITE`** operation.

3.1 The Basic Types

3.1-A The Z Version:

Types in Z seem to be rather informal; we merely declare, for example, that we want a type BYTE. We can declare sets, sequences, tuples, bags, relations, mappings, and functions. We can state whether functions are partial or total and (with a lambda expression) how they are computed. For our purposes we need the types

```
BYTE

FILE ≡ seq BYTE
```

We’ll also need the naturals, but these are primitive. A constant **ZERO** of type **BYTE** is required, but there seems to be no need to explicitly define it.

3.1-B The Gypsy Version:

The typing mechanism in Gypsy is more formal and more restrictive. Basic types such as **INTEGER** and **BOOLEAN** are available as are the static type compositions arrays and records and the dynamic type compositions of sets, sequences, and mapping. Gypsy also has buffer types for communication between concurrent processes. The Gypsy analog of the Z types above is:

```
type BYTE = pending;

type BYTE_SEQ = sequence of byte;

const ZERO: byte := pending;

type FILE = byte_seq;
```

The keyword **pending** in Gypsy is a conceptual place-holder which makes no commitment to the ultimate implementation.

Defining the natural numbers as a type in Gypsy is awkward. Whereas the integers are unbounded in Gypsy, there is no satisfactory way to specify an unbounded subset of them. The naturals are characterized in our specification as the collection of integers between 0 and some unspecified constant. The lemma (axiom) **MAX_NAT_POSITIVE** guarantees that this range is nonempty.

```
const MAX_NAT : integer := pending;

lemma MAX_NAT_POSITIVE =
      MAX_NAT > 0;

type NATURAL = integer [0..MAX_NAT];
```

This characterization of the naturals in Gypsy is clumsy. It is likely that for any language which is mechanically processed (as Gypsy is) there will be interesting concepts which cannot be formalized conveniently within a fixed notational framework. A language such as Z which is more freely extensible seems to have a distinct advantage in this regard. On the other hand, many would argue that the desire to include too much in a formal language is exactly the cause of complexity and inelegance in languages such as PL/I and ADA. [10]

3.2 Writing Files

3.2-A The Z Version:

The operation of writing a file is defined in the Z specification by the schema:

writeFILE

file, file' : FILE
offset? : $\mathbf{N}$
data? : seq BYTE

$file' = zero_{offset?} \oplus file \oplus (data? \circ pred^{offset?})$

where $zero_k = (\lambda n{:}\mathbf{N} \mid 1 \le n \le k \bullet ZERO)$

The uninitiated may find this specification rather daunting. Actually, it is quite simple once the meaning of the operators is understood.

$\mathbf{zero_k}$ is a sequence of length **k** all of whose bytes are **ZERO**. ⊕ is the function overriding operator; **f⊕g(x)** equals **g(x)** unless **g(x)** is undefined, in which case it equals **f(x)**. **pred** is the predecessor function. The specification states that to determine the value of any byte in the written file one must look first at the written data, then at the previous contents of the file, and finally to **ZERO**.

Notice again the very nonprocedural nature of the specification. There is no prescription *how* the final value of file is obtained, only of *what* the final value must be.

3.2-B The Gypsy Version:

It is possible in Gypsy to state the specification corresponding to the Z schema **writeFILE** in a very abstract fashion essentially as a functional relationship between the input and output values of the file. This might be expressed as:

```
function WRITE_TO_SEQUENCE (in_file : file;
                             offset : natural;
                               data : byte_seq) : byte_seq =
begin
   exit (result =
            if (offset le size (in_file))
               then in_file [1..offset-1] @ data
               else in_file @ n_zeros ([offset - size (in_file)] + 1)
                            @ data
            fi);
end; {write_to_sequence}
```

```
function N_ZEROS (n: integer): byte_seq =
begin
   exit (  result
         = if n le 0
       then null (byte_seq)
       else zero :> n_zeros (n - 1)
           fi);
   pending;
end; {n_zeros}
```

The Gypsy version is somewhat more verbose but quite similar in spirit to the Z version. Preference and experience determines which is more daunting. Notice the need in Gypsy to declare the auxiliary function **`N_ZEROS`** comparable to the Z construct $\mathbf{zero}_k$.

The Gypsy function **`WRITE_TO_SEQUENCE`** defines the desired input/output relation of the **`WRITE`** operation, but it is not the operation itself. It is natural in Gypsy to characterize the operation itself as a function or procedure and use the specification function **`WRITE_TO_SEQUENCE`** to state a constraint on its behavior.[4]

```
procedure WRITE_FILE (var fl : file;
                      offset : natural;
                        data : byte_seq) =
begin
   exit fl = write_to_sequence (fl', offset, data);
   pending;
end; {write_file}
```

This has essentially the same content as the Z schema, where the **`exit`** specification gives a postcondition of the routine analogous to the Z schema observation. The presence of the keyword **`pending`** in place of the procedure body indicates that no commitment is currently made to an implementation. Any implementation supplied later must satisfy the specification.

3.3 File Storage

3.3-A The Z Version:

In the file system we access a file via its *file_id*, a number supplied by the system when the file is created. This implies a mapping from file_id to files specified in Z by a new type **`FID`** and a schema.

```
___SS________________________________
|    fstore : FID ⇸ FILE
|____________________________________
```

A concept that recurs in several operations on stored files is the notion of accessing the file within the store. This is expressed in the following *framing* schema:

[4]Primed variables such as **`fl'`** represent the *input* values of variable parameters in Gypsy as opposed to the *output* values in Z.

___ϕSS___________________

SS, SS'
file, file' : FILE
fid : FID

____

file = fstore (fid)
$\text{fstore}' = \text{fstore} \oplus \{\text{fid} \rightarrow \text{file}'\}$

____

In traditional mathematical parlance this schema might translate as: *let fstore be a storage system in which file is associated with fid*. Notice that we specify explicitly how **`file`** is computed; thus in an expansion of this schema, we could replace all occurences of **`file`** by **`fstore (fid)`**. This notion is called *hiding* of the variable in Z.

Given the framing schema ϕ**SS**, the notion of writing a file in the storage system can be expressed in quite a terse fashion as:

___writeSS___________________

ϕSS
writeFILE

____

If desired, the expansion of this schema can be obtained by conjoining its constituent schemata. Common variables and observations are recorded only once. The result is

___writeSS___________________

SS, SS'
fid : FID
file, file' : FILE
offset? : $\mathbf{N}$
data? : seq BYTE

____

file = fstore (fid)
$\text{file}' = \text{zero}_{\text{offset?}} \oplus \text{file} \oplus (\text{data?} \circ \text{pred}^{\text{offset?}})$
$\text{fstore}' = \text{fstore} \oplus \{\text{fid} \rightarrow \text{file}'\}$

where $\text{zero}_k = (\lambda n{:}\mathbf{N} \mid 1 \le n \le k \bullet \text{ZERO})$

____

which may be simplified using any of the rules of predicate calculus.

3.3-B The Gypsy Version:

These same concepts may be expressed straightforwardly in Gypsy. The types we need are declared as follows:

```
type FILE_ID = pending;

type STORAGE_SYSTEM = mapping from file_id to file;
```

The functionality of **`writeSS`** is expressed in the following Gypsy procedure:

```
procedure WRITE_STORED_FILE (var SS : storage_system;
                                 FID : file_id;
                                data : byte_seq;
                              offset : natural) =
begin
   entry FID in domain (SS');
   exit  SS = SS' with (into [FID]
                   := write_to_sequence (SS'[FID], offset, data));
   write_file (SS[FID], offset, data);
end; {write_stored_file}
```

Notice that this procedure makes use of the earlier version just as the Z schema made use of its predecessor. The specification is the exact analog of that for **`WRITE_FILE`**, with the change made to a component of the file structure rather than to the file in isolation. This is the Gypsy counterpart of the Z notion of "hiding" discussed above. The key difference between the Gypsy and Z versions is that in the Gypsy code we have procedural abstraction rather than the schema abstraction of Z.

In this case we give a body to the procedure rather than leaving it **`pending`**. It seemed natural to do so for two reasons. It illustrates that **`WRITE_STORED_FILE`** is merely a specific instance of **`WRITE_FILE`**, where the file var parameter is obtained via indirection through its file_id. Also it allows us to *prove* the correctness of this procedure assuming the correctness of **`WRITE_FILE`**.

There is a subtlety in the Z specification which becomes more explicit in the Gypsy version. In the Z version, **`fstore`** is declared as a partial function; the observation **`file = fstore (fid)`** assures that it is defined at **`fid`**. This is stated explicitly in the entry specification on the Gypsy routine.

3.4 Sequential Access to Files

The next step in the development is to add the notion of sequential access to files via *channels*. A channel records an association between a file id and a current position in the file. Sequential access in the file commences from that position.

3.4-A The Z Version:

This association is made in Z with the schema:

```
___CHAN_____________________________
|   fid : FID
|   posn : N
|___________________________________
```

An additional important property of channels is that the **`fid`** of the channel never changes, expressed by:

```
___ΔCHAN_______________________________
|   CHAN, CHAN’
|_________________
|
|   fid’ = fid
|______________________________________
```

The operation of writing a file via a channel makes use of the previous schemata **`writeSS`** and Δ**`CHAN`** along with some observations to characterize the result.

```
___writeCHAN___________________________
|   writeSS
|   ΔCHAN
|_________________
|
|   offset? = posn
|   posn’ = posn + #data?
|______________________________________
```

Here the # operator returns the length of its argument. Notice that the parameter **`offset?`** to schema **`writeSS`** is supplied by the **`posn`** component of the channel.

Finally, we wish to add a named system of channels for performing sequential access. We add the data type **`CID`** of channel ids and a mapping from channel ids to channels.

```
___CS__________________________________
|   cstore: CID ⇸ CHAN
|______________________________________
```

We need also a schema describing the writing of a channel accessed via the channel system. This is merely an instance of the **`writeCHAN`** schema with **`posn`** supplied from the channel store.

```
___writeCS_____________________________
|   CS, CS’
|   writeCHAN
|_________________
|
|   posn = cstore (fid)
|______________________________________
```

3.4-B The Gypsy Version:

The most natural way in Gypsy of associating two dissimilar pieces of data is a record structure. We could have defined a **`CHANNEL`** record type of two fields. In the **`writeCHAN`** operation, this would be convenient. However, looking ahead to the use we’ll make of channels, it seemed that this way of structuring would be inconvenient.

This illustrates a characteristic difference between Gypsy and Z. In Z, the association of data within a schema has no connotations for an implementation structuring. Individual pieces of data can be aggregated into various different schemas. In Gypsy, on the other hand, associating data items into a structure, a record for example, makes it very difficult to re-associate those data items differently at a later point.

Our declarations and the definition of the operation for writing a file via a channel are specified as follows:

```
type POSITION = natural;

type CHANNEL_ID = file_id;

procedure WRITE_CHANNEL (var SS : storage_system;
                           data : byte_seq;
                        chan_id : channel_id;
                            var posn : position) =
begin
   entry chan_id in domain (SS);
   exit  SS = SS’ with ([chan_id]
                     := write_to_sequence (SS[chan_id], posn’, data))
       & posn = posn’ + size (data);
   write_stored_file (SS, chan_id, data, posn);
   posn := posn + size (data);
end; {read_channel}
```

Because we did not want to create a single data structure representing the channel, it was necessary to pass the channel_id and position as separate parameters. This has an associated benefit of guaranteeing syntactically that the channel_id parameter could not be altered by the procedure invocation as called for in the Z schema Δ**CHAN** since it is a *const* rather than a *var* parameter. If the channel id and position parameters had been fields in a record parameter to this routine, it would have been more difficult to assert that **WRITE_CHANNEL** does not alter the channel id.

We now define the channel system as the following mapping:

```
type CHANNEL_SYSTEM = mapping from channel_id to position;
```

The operation which allows us to write a file sequentially using the channel system is coded in Gypsy as follows:

```
procedure WRITE_CS ( var SS : storage_system;
                     var CS : channel_system;
                    chan_id : channel_id;
                       data : byte_seq) =
begin
   entry  chan_id in domain (CS)
        & chan_id in domain (SS);
   exit   SS = SS’ with ([chan_id]
              := write_to_sequence (SS’[chan_id], CS’[chan_id], data))
        & CS = CS’ with ([chan_id] := CS’[chan_id] + size (data));
   write_channel (SS, data, chan_id, CS[chan_id]);
end; {write_CS}
```

It is necessary to pass in both the storage system and channel system since both the file and the current position are updated by the **WRITE_CS** operation. Notice also that we need to assure in the entry specification that the channel_id is a proper file_id in the file system. We’ll address this issue again in the next section.

3.5 The Access System

The last component of the specification we'll consider is the access system. The access system is merely the combination of the storage and the channel system. However, we want to assure that no channel contains a file id for which there is no associated file in the storage system.

3.5-A The Z Version:

This is expressed in Z by the following schema.

```
___AS_________________________________
|    SS
|    CS
|__________________
|
|    ran (fid∘cstore) ⊆ dom fstore
|_____________________________________
```

The observation in this schema expresses an invariant which must be preserved by every operation on the access system. Since the schema abstraction is nonprocedural, the invariant is simply inherited by every schema which uses **AS**.

The write operation using the access system is specified using the framing schema

```
___φAS________________________________
|    AS, AS'
|    ΔCHAN
|    cid : CID
|__________________
|
|    CHAN = cstore cid
|    cstore' = cstore ⊕ {cid → CHAN'}
|_____________________________________
```

and the following combination

```
___writeAS____________________________
|    φAS
|    writeCHAN
|_____________________________________
```

3.5-B The Gypsy Version:

The desire to associate an invariant with a collection of data structures leads naturally in Gypsy to the abstract data typing facility. In this case the data type represents the aggregate of the storage system and the channel system.

```
type ACCESS_SYSTEM <SS, set_SS, ...> =
   begin
      AS: record (SS: storage_system;
                  CS: channel_system);
      HOLD domain (AS.CS) sub domain (AS.SS);
   end;
```

The type **ACCESS_SYSTEM** is a Gypsy abstract data type. The abstract typing mechanism in Gypsy serves two distinct purposes: to hide the implementation of a type and to permit the association of an invariant with the type in the form of the **HOLD** specification. The *access control list* including **SS**, **set_SS**, and possibly others gives the list of routines which are permitted access to the concrete (record) structure of the type. Each of these must be proved to maintain the invariant.

The use of Gypsy data abstraction for our example is somewhat unfortunate because we are really concerned only with maintaining the invariant; the data hiding aspect of abstract typing is primarily a nuisance in the current context. Since we will want to access the various components of the access system, it is necessary to write functions which will permit us to access and set components. For the storage system component, such functions are:

```
function SS (AS: access_system): storage_system =
begin
   cexit result = AS.SS;
   result := AS.SS;
end; {SS}

procedure set_SS (var AS : access_system;
                      SS : storage_system) =
begin
   cexit AS = AS’ with (.SS := SS);
   AS.SS := SS;
end; {set_SS}
```

We would have similar functions for the channel system component of the abstract type. It is also necessary to define a special function which characterizes equality for the abstract type.

It is syntactically disallowed for any routine to refer to the concrete structure of the abstract type except those routines mentioned on the access control list. Even these cannot refer to the concrete structure in their abstract external specifications (**entry** and **exit**). The **centry** and **cexit** specifications of these routines may refer to the structure, but they are visible only in proof contexts in which the concrete structure of the type is visible. Thus **SS** and **set_SS** are abstract accessors which must be used in most contexts in place of direct access to the **SS** component of the record structure.

The write operation using the access system then becomes

```
procedure WRITE_AS ( var AS : access_system;
                    chan_id : channel_id;
                       data : byte_seq) =
begin
   entry  chan_id in domain (CS (AS));
   exit  SS(AS) = SS(AS) with ([chan_id]
                := write_to_sequence (SS(AS)[chan_id],
                                      CS(AS)[chan_id], data))
       & CS(AS) = CS(AS) with ([chan_id]
                                   := CS(AS)[chan_id] + size (data));
   write_channel (SS(AS), data, chan_id, CS(AS)[chan_id]);
end; {write_AS}
```

The astute observer will have already noted that this is exactly the definition of **`WRITE_CS`** with adjustments made for the abstract data typing and the combination of **`CS`** and **`SS`** parameters into one structure.

Though we stop our exposition of the specifications here, the interested reader is invited to investigate the complete specifications. The Z version of the full spec is described in [18]. Our Gypsy version is available upon request.

4. Comparing the Specifications

Our investigation of the UNIX Filing System example has highlighted various features of the two specification languages. Though the resulting specifications are superficially quite different, we have attempted to point out the underlying similarities. We would aver that both specification languages can result in elegant readable specifications if used with skill and care. There are distinct differences, though, which are worthy of note.

4.1 Expressiveness of the Languages

The fact that Gypsy is an implemented language means that there are certain constraints upon the expressiveness of the language imposed by the parser. Gypsy contains the full first order predicate calculus and the ability to define functions recursively. In addition, there is an extensive collection of data types including sets, sequences, and mappings. However, we have seen that it is sometimes awkward to express certain concepts—the natural numbers are a good example—in a satisfactory fashion. Also, the lack of polymorphism in Gypsy means that it is often necessary to write very similar functions to perform analogous operations on, say, sequences of integers and sequences of Booleans. Many concepts which might be desirable from a programming standpoint—pointers, floating point, global variables, functions with side effects—are explicitly excluded because of the difficulties they present for verification.

Z suffers from no such lack of expressiveness. In addition to the huge notational variety suggested by the language designers [9], the writer of Z specifications is free to use or invent notation at will. This gives the Z user freedom to write the cleanest specifications possible.

On the other hand, the free and easy use of notation in Z may have some disadvantages. The semantics of Z is inherited from mathematics and in that respect well defined. However, this is an extremely powerful underlying theory. There is certainly no way to insure that Z specifications are realizable or even consistent. The same can be said of Gypsy specifications, though the procedural nature of the language imposes a bias toward constructive specifications. Z specs are often highly non-constructive.

Also, much of the readability of a Z spec derives from various notational conventions: the suffixes distinguishing input and output variables, for example. Since these are not enforced by a language processor it is possible to violate them quite easily.

4.2 Structuring of the Specifications

The primary structuring concept in Z is the schema abstraction; in Gypsy it is procedural abstraction. Either permits a well structured development style and a clean modular specification. Z seems to win for sheer brevity and abstractness of the resulting specifications.

Much of the verbosity of the Gypsy specs comes from Gypsy's proscription of non-local referencing. This requires that all data structures accessed within a module be either local or passed in as parameters and this tends to clutter the procedure header. However, this has the strong advantage that Gypsy modules can be understood in isolation from their calling environments and that the effects of a Gypsy procedure are very strictly constrained. A constraint on the language called the *Independence Principle* assures that any module is analyzable/provable with regard only to its own code and the external specifications of any routines it calls.

Fully understanding a Z spec may require expanding all of the schemata in the tree of schema definitions defining it. This could be formidable indeed. The use of schema naming conventions seems to make this seldom necessary. However, again this requires that the specifier follow the standard conventions.

Data structuring in Z is much less constraining than in Gypsy. Consider our formalization of the Unix *access system*. By encapsulating the storage system and channel system into a single abstract data type, we conceptually bind them together. In a later context (not discussed above) it is convenient to consider a combination of the storage system with some other combinations of systems (not including the channel system). In Z, this means simply including the appropriate schemata; including one schema within another does not "hide" it from any future uses. In Gypsy, there is a conceptual structural mismatch arising from considering the "same" system component as a piece of two different aggregations.

4.3 Procedural vs. Nonprocedural

The debate over the relative merits of procedural versus non-procedural specifications at times takes on an almost religious fervor. Suffice it to say that Z is highly non-procedural. Gypsy specifications can be non-procedural though this is typically not the most natural style. The advantage of non-procedural specifications is that they are largely implementation independent. This is evident from the Z file system example. The danger of non-procedural specifications is the lack of assurance that they are realizable.

The ability to supply bodies to our Gypsy procedures proved seductive and insofar as these procedure bodies are considered as part of the specification, they constrain allowable implementations. On the other hand, the advantages of executable specifications for rapid prototyping have often been cited. Gypsy specifications are not executable because of the current lack of an interpreter for the language. It would not be difficult to supply one, however.[5] Also, some aspects of the Gypsy specification language are intended for *run-time validation*, the evaluation of specification expressions during program execution. This allows the checking of specifications very directly against the run-time behavior of a system.

4.4 Amenability to Code Level Specification

A difference in these specification styles which may be particularly relevant to secure system development efforts is the applicability to specifications at or near the code level. One of the increasing demands upon verification technology alluded to earlier is the demand to close the gap between the specification level and the machine code implementation of the system running on actual hardware. This is evident particularly in the requirements of the "beyond A1" certification level of DoD’s *Trusted Computer Systems Evaluation Criteria* [3].

Z has been used for specifying some "real" software systems of impressive size including one system of over 80,000 lines of code at IBM-Hursley. There is no reason why a specification in Z cannot be as detailed and as near the code level as is required. There is also ongoing research into refining Z specifications into code in a guarded command language [17]. Presumably this could then be translated into C or other suitable implementation language.

[5]It might be possible to supply some such notion for Z as well. Logic programming gives a procedural as well as a declarative interpretation to logical formulae. However, Z is a much richer language than typical logic programming languages.

Gypsy has been used for code level specification and verification on several projects. [21, 19] For these projects Gypsy was used as the implementation language and mechanically translated into Bliss which was then compiled. [20] Most current uses of Gypsy in the development of secure system applications, however, have been for specification at the design level. There is currently no Gypsy compiler available except a prototype verified compiler for a very small subset of the language [25]. The result is that Gypsy design level specifications are translated by hand into C or some other suitable implementation language, an error-prone process [26].

Arguably, Gypsy has an advantage over Z in this process in that there is a clearer mapping from procedural Gypsy code to an implementation than from a non-procedural Z specification to an implementation. However, it may also be that the procedural nature of a Gypsy specification obscures rather than clarifies the mapping if the implementation is structurally quite different from the spec.

4.5 Mechanical Support

The clearest distinction between Gypsy and Z is in the area of mechanical support for language processing. There currently seems to be no language processing capability for Z. The Gypsy Verification Environment (GVE), on the other hand, is a mature and well integrated collection of software tools for developing and processing Gypsy programs and specifications. These tools include a parser, database system, verification condition generator, interactive proof checker, and algebraic simplifier.

A mechanical parser is particularly beneficial from the point of view of writing consistent specifications. We noticed in studying the Z Unix File System specification [18] that there was a least one schema (**`createCS`**) which is referenced but never defined.[6] This sort of oversight is very easy to eliminate with mechanical parsing.

4.6 Proofs about the Specifications

For both languages, it is possible to do proofs about the specifications. For Z, this follows from the fact that the specification language in some senses just *is* elementary mathematics. Users interested in doing proofs will find themselves on the safe and sure ground of elementary mathematics. Proof, however, does not seem to be a high priority for Z users. Possibly this is because the focus of Z use is on *specification* of software systems, not on formal verification which tends to focus on proving the conformity of specifications and code.

[6]Interestingly enough, this is actually evident from the index of schemas and components given by the authors. Schema **`createCS`** is listed as being used within schema **`create`**; there is no separate entry for the *definition* of **`createCS`**.

The Gypsy system is very heavily oriented toward proofs of correctness. An overriding design criterion for the language was that every construct have associated proof rules. The verification condition generator processes programs annotated with assertions to generate verification conditions (VC's) adequate to assure the conformity of code and specifications. These verification conditions are conjectures which can be proved using the Gypsy interactive theorem prover. It is also possible to state and prove *lemmas*.

The need to resort to the process of verification condition generation for Gypsy programs is due to the procedural nature of the language. It is sometimes argued that the process of VC generation obfuscates the relation between specifications and code. The VC's often bear little obvious relation to the code. However, this seems to be a necessary price for having procedural constructs in the language. It is possible to reason about procedural programs directly with respect to a formal semantics, but it is much more difficult to do so [25].

5. Conclusions

We have compared and contrasted two specification languages—Gypsy and Z—in light of a common example. Each provided some obvious advantages and disadvantages.

Z allows the construction of very clear and elegant specifications. It has been used with good results in specifying large software systems. The principle failings of the language and its current usage seem to be the following.

1. The expressive freedom allowed by the language can, if abused, easily result in specifications which are either not satisfiable or for which there is no efficient implementation.
2. Because of the highly non-procedural character of specifications in Z, there may be no clear mapping from specification to implementation. Thus it might be very difficult to construct a believable specification to code correspondence argument.
3. The greatest failing of Z currently seems to be the lack of mechanical support for language processing. Inconsistencies and gaps in the specifications could be easily eliminated by a parser.

Gypsy is a combined specification and programming language with extensive software support. The following comments can be made about Gypsy and its implementation.

1. Some common mathematical notions are difficult to express in Gypsy. We noted the natural number data type as an example; because of the absence of pointers, trees are also awkward to express. However, it is not quite accurate to say that Gypsy is uniformly less expressive than Z. It is unclear, for example, how difficult it would be to specify in Z concurrent programs which Gypsy allows.[7]
2. The mechanisms of abstraction in Gypsy—procedural and data abstraction—are less flexible

[7]There is apparently ongoing research on specifying concurrent programs in Z based on the approach of [12].

than schema abstraction. In particular, it is difficult to associate components of a system into various *different* aggregations.

3. Schema based specifications tend to be more succinct and abstract than the procedural specifications of Gypsy. This can be interpreted as implying that the procedural nature of Gypsy specifications imposes unnecessary constraints on an implementation.

Our experience in comparing Gypsy and Z leads us to believe that the relative strengths of the two specification languages are in fact quite complementary. The major failing of Z—the lack of mechanized language support—is also the easiest to remedy. The lessons learned in the development of the Gypsy Verification Environment could serve as a model for the development of a mechanical support environment for Z.

Because of its procedural constructs and strong mechanical support for proofs about programs, Gypsy will likely continue to have the edge over a language like Z in secure system development efforts. But the lessons gained by comparing these very different specification paradigms may inform future changes and improvements in both languages and their support environments.

Acknowledgments: Thanks to Matt Kaufmann and to the reviewers for some quite insightful comments on an earlier version. This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

References

1. W.E. Boebert, W.D. Young, R.Y. Kain, S.A. Hansohn. Secure ADA Target: Issues, System Design, and Verification. *Proc. Symposium on Security and Privacy*, IEEE, 1985.

2. M. Cheheyl, M. Gasser, G. Huff, J. Millen. "Verifying Security". *ACM Computing Surveys 13*, 3 (September 1981), 279-340.

3. Department of Defense. *Trusted Computer Systems Evaluation Criteria*. DOD 5200.28-STD, December, 1985.

4. Bill Flinn and Ib Holm Sorensen. CAVIAR: A Case Study in Specification. In Ian Hayes, Ed., *Specification Case Studies*, Prentice-Hall, Englewood Cliffs, N.J., 1987, pp. 141-188.

5. D.I. Good. SCOMP Trusted Processes. ICSCA Internal Note 138, The University of Texas at Austin.

6. D.I. Good, R.L. Akers, L.M. Smith. Report on Gypsy 2.05. Tech. Rept. ICSCA-CMP-48, Institute for Computer Science and Computing Applications, The University of Texas at Austin, February, 1986.

7. D.I. Good, B.L. Divito, M.K. Smith. Using The Gypsy Methodology. Institute for Computing Science, University of Texas at Austin, June, 1984.

8. D.I. Good, A.E. Siebert, L.M. Smith. Message Flow Modulator Final Report. Tech. Rept. ICSCA-CMP-34, Institute for Computing Science, University of Texas at Austin, December, 1982.

9. Ian Hayes (editor). *Specification Case Studies.* Prentice-Hall, Englewood Cliffs, N.J., 1987.

10. C.A.R. Hoare. "The Emperor’s Old Clothes: 1980 Turing Award Lecture". *Communications of the ACM 24*, 2 (February 1981), 75-83.

11. K. Jensen and N. Wirth. *Pascal User Manual and Report.* Springer-Verlag, 1974.

12. C.B. Jones. "Tentative Steps Toward a Development Method for Interfering Programs". *ACM Trans. on Programming Languages and Systems 5*, 4 (October 1983), 596-619.

13. Matt Kaufmann and W.D. Young. Comparing Specification Paradigms for Secure Systems: Gypsy and the Boyer-Moore Logic. Proceedings of the 10th National Computer Security Conference, National Bureau of Standards, September, 1987.

14. J. Keeton-Williams, S.R. Ames, B.A. Hartman, and R.C. Tyler. Verification of the ACCAT-Guard Downgrade Trusted Process. Tech. Rept. NTR-8463, The Mitre Corporation, Bedford, MA., 1982.

15. Richard Kemmerer. Verification Assessment Study Final Report. In 5 volumes, unpublished.

16. Carl E. Landwehr. "The Best Available Technologies for Computer Security". *IEEE Computer 16*, 7 (July 1983), 86-100.

17. C. Morgan, K. Robinson, P. Gardiner. On the Refinement Calculus. Draft, July 1988.

18. Carroll Morgan and Bernard Sufrin. Specification of the UNIX Filing System. In Ian Hayes, Ed., *Specification Case Studies*, Prentice-Hall, Englewood Cliffs, N.J., 1987, pp. 91-140.

19. A.E. Siebert and D.I. Good. General Message Flow Modulator. Tech. Rept. ICSCA-CMP-42, Institute for Computing Science, University of Texas at Austin, March, 1984.

20. L.M. Smith. Compiling from the Gypsy Verification Environment. Tech. Rept. ICSCA-CMP-20, Institute for Computing Science, The University of Texas at Austin, August, 1980.

21. M.K. Smith, A. Siebert, B. Divito, and D. Good. "A Verified Encrypted Packet Interface". *Software Engineering Notes 6*, 3 (July 1981).

22. J.M. Spivey. *Understanding Z: a Specification Language and its Formal Semantics.* Cambridge University Press, 1988.

23. Bernard Sufrin. Towards a Formal Specification of the ICL Data Dictionary. In Ian Hayes, Ed., *Specification Case Studies*, Prentice-Hall, Englewood Cliffs, N.J., 1987, pp. 189-217.

24. J.C.P. Woodcock. Structuring Specifications: Notes on the Schema Notation. Oxford University Computing Laboratory, August, 1987.

25. W.D. Young. A Verified Code Generator for a Subset of Gypsy. Tech. Rept. CLI-33, CLInc, November, 1988.

26. W.D. Young, J. McHugh. Coding for a Believable Specification to Implementation Mapping. Proceedings of the 1987 Symposium on Security and Privacy, IEEE, 1987.

Table of Contents