Supporting the DSL Spectrum[1]

David S. Wile
Teknowledge Corp., Marina del Rey, CA 90292, USA

A language tailored to the problem domain can focus on its idioms and jargon, avoiding clumsy, overly general constructs needed to support general-purpose language. The leverage provided by DSLs over conventional programming languages is often extreme; application engineers may specify as little as 2% of the code that one would need to program the same thing in a conventional programming language! But commitment to a DSL approach can be rather expensive.

It is often difficult to know when to invest in exactly how much infrastructure support for a product or product family. All of the concerns that are germane to general-purpose programming language design and support may become important in the support of a specific DSL. At the same time, there is a wide spectrum of approaches to providing DSL support. This paper relates the various DSL design approaches to alternatives for tool support, providing a kind of "DSL tool support selection framework," indicating where one might expect to need to invest heavily to obtain adequate support and illustrating the spectrum of tradeoffs and situations in which each is appropriate.

Keywords: domain-specific language, language extension, COTS adoption, XML, general-purpose language

1. Where to Begin?

The design and use of domain-specific languages can be seen as the natural culmination of the use of domain-specific techniques for program and product development. By focusing on a problem domain's idioms and jargon, DSLs avoid the notational "noise" required when using overly general constructs of a general-purpose language to express the same thing. *Moreover, DSLs are not necessarily programming languages: they are languages tailored to express something about the solution to a problem.* Indeed for some uses of DSLs there is really no comparison with an equivalent "program." For example, music notation constitutes a DSL for music; it is nearly irrelevant that modern computers can play the "program" represented by the music. Even before MIDI existed (allowing computers of various sorts to play the music), editors existed for this DSL, so printers could render the compositions without humans having to "retype" them on scored music sheets each time they changed them.

It is generally very clear that DSL usage is very effective for the users, the practitioners in the problem domain who write specifications in the languages. With a language we designed for NATO message parsing and type-checking[2], we experienced a 50 to 1 improvement in lines of generated code vs. lines of specification [Balzer94]. Additionally, confidence in the correctness of the specifications is greatly improved when all the extra syntactic noise is eliminated from the specifications.

However, commitment to a DSL approach can be rather expensive, depending on the level of infrastructure support provided for a product or product family. All of the concerns that are germane to general-purpose programming language design and support may become important in the support of a specific DSL. These concerns include designing and specifying the actual syntax of the DSL itself; designing abstract or internal representations; compilation concerns, such as parsing, type checking, and

[1] This work was sponsored by DARPA contract numbers F30602-96-2-0224 and F30602-00-C-0200.

[2] NATO has a standard for messages transmitted between all kinds of vehicles and processing stations, from tanks, to planes, to command centers. These messages consist of related individual lines of text; each line of text comprises several "fields." The structure of the lines in terms of fields, and of individual message types as sequences of particular typed lines, was captured for the experiment described. A set of message types was then automatically translated into Ada to form the basis for the comparison mentioned in the text.

code generation; the design of other analyzers, such as usage graphs; means for debugging DSL "programs," often entailing an "unparsing" activity to map back out of the internal representation; support for persistence, versions and multiple authors; and often mechanisms for simulating or executing DSL programs.

To add to the confusion, there is a wide spectrum of approaches for providing DSL support. Beyond ad hoc approaches — rarely worthwhile investments — there are traditional methods using programming language development tool technology, such as LEX/YACC-based derivatives [Aho86, Sharnoff01], programming environment generator designs such as [Reps88, Klint93], and other language support environments [van den Brand01, Mernik00a, Wile93]. A fairly popular technique is to extend existing languages with abstract data types and operators, or object classes and methods, characterizing the domain so thoroughly that application engineers can become programmers without learning too much of the underlying languages [Hudak00]. A related technique of recent interest in the research community is the use of so-called "generic programming" to provide higher-level abstractions whose mappings into the language affect DSL-like constructs [Jeuring95]. Another related technique is to capture problem domain idioms using database schema and presentation mechanisms or spreadsheet templates. And, finally, XML-based approaches are becoming popular, along with tools for mapping over the abstract syntax captured in the XML representation [Attalli01, XMLSoftware]. We at Teknowledge are pursuing yet another approach, the use of a graphical interface in which "styles," comprising icons and different kinds of arrows, can be mixed with textual and graphical attributes to form a kind of hybrid DSL technology [Goldman99].[3]

In the following, advantages to DSL approaches are first discussed in Section 2. Then, in Section 3, some of the various DSL design approaches just mentioned will be elaborated in more detail. In order to compare the various advantages of the different approaches, Section 4 presents a brief discussion of the different issues that arise when considering adopting a particular technology. Section 5 compares the various approaches with respect to how they handle these issues. With each approach I will indicate where one might expect to need to invest heavily to obtain adequate support and try to illustrate the spectrum of tradeoffs and alternative approaches that might be pursued.

2. DSL Advantages

So, just what are the advantages of using a DSL, specifically?

First, with conciseness come a lot of advantages. Foremost is comprehensibility: one can read more at once and relate pieces whose connections may be completely lost in formal noise when couched in the terms of a more general-purpose language. As an aside, it is worth mentioning that there are many general-purpose problem (solution) *specification* languages; e.g. Prolog [Bowen01] has a bit of the flavor of one, where the evaluation model is somewhat far from the von Neuman machine model. Some of these languages are not even "evaluable": they may allow expressing solutions in second order predicate calculus [Bowen01]. Even these much higher-level languages are often extremely clumsy compared with a language tailored to the domain. To return to the thread, a second benefit to conciseness is ease of writing and, consequently, introduction of fewer clerical mistakes. Finally, conciseness leads to greater productivity. The common rubric that claims that the number of lines of code per programmer per day is nearly constant, independent of the language used, *probably* carries over into other specification domains.[4]

A second broad category of advantages accrues due to the fact that the natural internal representations for the linguistic constructs, their "abstract syntax" taken in a very broad sense, are much more attuned to the kinds of analysis and processing that will go on in designing analyzers, simulators, animators, and translators for the languages. This has two implications: (1) the analyzers can be written more easily and directly than when decoding a more general-purpose representation and (2) a more

[3] In fact, we are going one step further and relating these styles to specific knowledge representation "ontologies," but I will not go into that further in this paper [Tallis01].

[4] But I admittedly have no proof or even anecdotal backup for this conjecture.

thorough-going job can be done of their implementation, since less time need be devoted to circumventing a clumsy original representation. In addition, the errors can more readily be related directly to the users in problem domain terms rather than in the terms of the supporting general-purpose language. In fact, this ability to develop ancillary tool support for the application engineer, tools tailored to the domain — perhaps meaningless outside the domain, represents a tremendous advantage over alternative approaches.

But of course, the single greatest advantage provided by the use of domain-specific languages is that the expert at solving the problem can write specifications rather than requiring a combined expert-programmer team to code. This is not to claim that the role of programmer becomes irrelevant; far from it. The overall programmer time investment is quite reduced indeed, but necessary for up-front infrastructure development. And in fact, the programmer skills required to provide the infrastructure are *more expensive*, requiring greater expertise than the normal program support staff. In some sense, the rest of the paper is devoted to illustrating just what these skills are and just how much or little of them might be needed when deciding how to provide support for the use of a domain-specific language.

3. DSL Approaches

As I mentioned above, the spectrum of approaches available for providing DSL support ranges from the development of infrastructure using ad hoc methods to language development environments, to language extension, through Common Off-The-Shelf (COTS) product usage — like Access, Excel, or PowerPoint — and bottoms out at interchange representations. Before discussing these approaches it is worthwhile to try to characterize what constitutes a DSL. And in order to do this, it will be illustrative to pick a particular (toy) problem domain to illustrate some of the issues.

So first, a problem domain. There are many problem domains that are "obviously" candidates for domain- specific languages, such as music composition and performance, inventory control, scheduling, accounting, signal processing, flight control, weather prediction, etc. The obvious nature of these for linguistic support comes from the fact that strong mathematical or otherwise formal languages have been used to describe problem solutions in them, and the experts already know — indeed have been trained in — these dialects. Other obvious candidates include computer-specific domains, such as compilation (YACC), regular expressions [Klarlund99], web computing [Cardelli99], cache coherence protocols [Chandra99], and device driver design [Thibault99]. However, other less obviously likely candidates have already been given DSL support: multimedia animation [Elliott99], music performance [Hudak95], NATO message processing [Balzer94, Kieburtz-94], naval ship formation movement analysis [Feather86], Interlocking in Railway Applications [van den Brand96], and census survey instrument definition [Wile00].

Unfortunately for our purposes here, all of these suffer from two defects: they require extensive domain understanding, too considerable to present here (even if I could) and they are somewhat large themselves. So, consider the admittedly contrived domain of "satellite ground

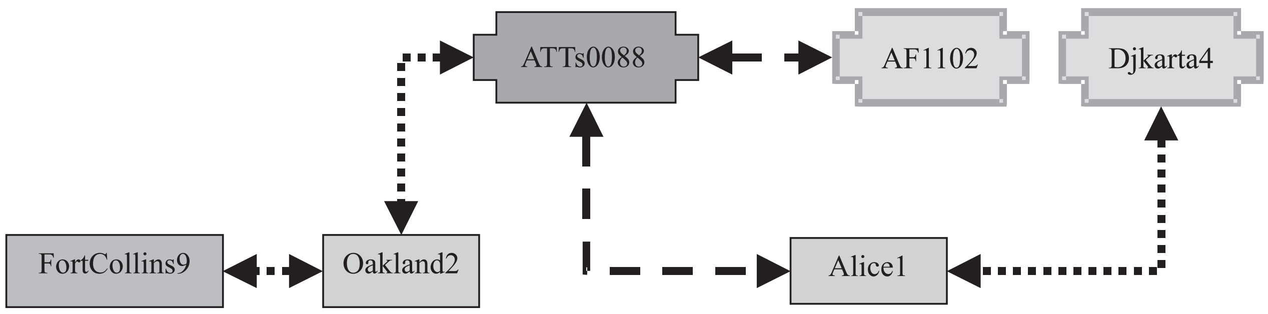


Fig. 1. Sattellite Communication System.

controllers" illustrated graphically in Figure 1.[5] Satellites are represented using cruciform boxes and ground stations are represented using simple rectangles. There are two kinds of satellites — communications satellites (darker, like *ATTs0088*) and probes (lighter) — and two kinds of ground stations — regular ones (lighter) and command processors (darker, like *FortCollins9*). There are communications links among satellites and ground stations indicated by dashed double-headed arrows. In some cases these are made and broken dynamically, as they travel into and out of the ranges of each others' receivers and transmitters as the satellites revolve about the earth. Here the dynamic links are represented with dashed connectors; the static ones with dotted connectors. The problem is to support the specification of the various satellites and ground stations — called "elements" from now on — in order to reason about such things as: when linkages can be established, whether the transmissions are appropriately secure, when they can be "reprogrammed," how much bandwidth is required, etc. Several aspects may require specification: elements' *structure* and properties, such as communication protocols; their *configuration* into a system; their interaction and interconnection *constraints*; and some aspects of their *behaviors*. These common elements of language design will be used in the following to evaluate the appropriateness of various approaches to providing DSL support. Also, in the following discussion the strengths of the approaches will generally be emphasized; later discussion will compare the strengths and weaknesses.

3.1. Full Language Design Approach

The domain-specific *language* approach to this example — with emphasis on developing a new specification language — might allow language users to produce something like the specification in Figure 2. The overall specification has a bit of the look-and-feel of a set of declarations in a programming language. Each element type has its own particular attributes that may be specified. For some reason the country owning the element — e.g. *Australia*, and possibly the corporate identity of the owner — e.g. *USA*'s *DoD* (Department of Defense), seemed important enough for it to be specified as the first atom(s) in the descriptions. Then comes the specific type of element, such as *surveillance controller*, and then a unique identifier or name. Other than ownership, this same information is conveyed in the diagram in Figure 1.

There are several features of this specification that deserve emphasis. Notice that everything has a *position* attribute and a *protocols* attribute. Since this example is contrived, it is difficult to say exactly what these protocols might be, but my intent is for them to refer to the security of the transmissions necessary to relay the data. After the common element specifications, each element is further described with syntax germane to the element's type, so, for example, some of the attributes used by ground stations are irrelevant to satellites — such as *Field of View* (see Figure 2) — and cannot be specified with them. (The field of view of the transmitter/receivers are given with the ground stations to reason about valid linkages.[6]) Notice also that position information does not include the altitude of ground stations nor communications satellites (presumably at a fixed altitude to stay in a stationary earth orbit). It is typical of DSLs to have specialized syntax for even conceptually "shared" fields.

One kind of analysis that will be important to perform on systems described using this language concerns policy analysis to determine which controllers can talk to which satellites, and through exactly which ground stations. Some of these paths are declared explicitly. Notice that there are three link statements after the elements are described in Figure 2; the intent here is to describe links that are predefined. When these occur with satellites, the satellites must actually be in a stationary earth orbit. Hence, the protocols that the ground stations affect, and the transformations in them that one can expect, are explicitly spelled out. There are two forms of these specifications used for the ground stations here: one says which protocols can be sent up and down by indicating the transformations of the input signals to the uplinks and the transformations to the output signals from the downlinks. The other form simply specifies

[5] The intent here is to appeal to a universal shallow understanding of this domain. Deeper knowledge of the domain will probably be a hindrance!

[6] Again, this is quite naive; a much more precise and differentiated set of coordinates and angles would probably be required.

```
USA surveillance controller FortCollins9
    Position: N40.5800 W105.0833
    Protocols: out secret S1
               in secret P8
    Hours: 7:00 am to 9:00 pm CST
USA ground station Oakland2
    Position: N37 48’ W122 16’
    Protocols: up secret P8
               down secret S1
               out {public S1, secret P8}
               in {secret S1, public S1}
    Field of view: 60
Australia ground station Alice1
    Position: 23° 48’ S 133° 53’ E
    Protocols: up {public P8 => private S1, public P8 => public S1, private S1 => private S1}
               down {private S1 => private S1}
    Field of view: 120
        *     *     *
USA communications satellite ATTs0088
    Corporation: ATT
    Position: N21 18’ W157 52’
    Multiplex 4096
    Rate: 100 MBS
    Protocols: Private P8
USDoD surveillance satellite AF1102
    At: 655446765 UT
    Position = [14:06.33992 hours x 27.9943 degrees x 75.66668 miles]
    Velocity = [ .2 hours x .3 degrees x - .00025 miles ] / hour
    Protocol: down secret P8
              up secret S1
    Sequencing:
    Transmitter: { on -Light-> off, off-receiver[data ready]-> on}
    Receiver: { on -noinput-> data ready, ...}
EU surveillance satellite Djkarta4
    Position = 6° 11’ S 106° 50’ E x 75.66668 miles
    Velocity = stationary
    Protocol: down private P8
              up public S1
    Sequencing =
    Camera: { on -Dark-> off, off-Light-> awaitTarget, . . .}
        *     *     *
Australia link Alice1 to Djkarta4
USA link FortCollins9 to Oakland2
USA link Oakland2 to ATTs0088
Compatibilities: public => { Australia, EU, USA }
                 secret => { Australia, EU }
                 private ATT => { Australia, EU, USA} ATT
```

Fig. 2. A Satellite Groundstation Configuration Specification.

what the input, output, up and down protocols are, and one assumes (unless otherwise constrained) that the cross product of the “in” with the “up” and the “down” with the “out” protocols are all possible.

This provision for syntactic sugar is common in DSLs. Notice as well that several forms of syntax have been provided for angles, latitudes, longitudes, etc. (e.g. “23° 48’S 133° 53’E” or “N21 18’W157 52’ ”). This is typical in DSLs, because the application engineers are generally accustomed to using different formats, e.g. from different instruments.

Notice also that there are aspects of *behavior* specified for satellites, such as initial position and velocity at a particular time and experiment sequencing information. Naturally, it would take a domain- specific interpreter to make sense of this information for analysis and simulation purposes.

The last lines of the specification describe some domain-specific *constraints* that specifications must satisfy: namely, when using public communication lines, all communications among elements owned by Australia, the EU and the USA are allowed. For some reason, even secret transmissions between Australian and EU elements are allowed. And, finally, private ATT communications are permitted among all three owners as well. This is a very concise way to say this and is very typical of the expressiveness of a DSL.

For now, take this sample specification as indicative of a rather small, but not insignificant language tailored to the domain of satellites. Next, we consider how one might implement the language support for reasoning about specifications in the language.

To provide support for this language as it was presented here, one would use traditional programming language development tool technology, such as LEX/YACC-based derivatives, or perhaps the Cornell Program Synthesizer. The latter entails defining an abstract syntax for the language, for which one gets a free syntax-directed editor for specifications in the language. One must also provide a concrete syntax for the language, matching the syntax given here, and sets of attribute grammar rules to effect various analyses and simulations that one might perform. For many situations, this is the appropriate choice to make. However, the programming expertise to deal with both the syntactic and semantic concerns is fairly expensive. Moreover, in some sense, the language design itself constitutes a heavy investment, and it has already been taken "for granted" here.

3.2. Language Extension Approaches

A less expensive alternative can be found in a broad category of alternatives loosely characterized as *extension* of an existing language to specialized constructs for the problem domain. Technically, the trick used is to refine the language with specific abstract data types and operators, or object classes and methods, characterizing the domain so thoroughly that application engineers can become programmers without learning too much of the underlying languages. They use the extended language — the original plus the refinements — to express their specifications and invoke their analyses and simulations. In the following, Java and Haskell will be used to illustrate the extension technique, chosen as current representatives of the OO programming approach and the declarative programming approach, respectively.

Java as a DSL. To approach our example, we might design the Java class specified in Figure 3a [Gosling96].[7] Based on these declarations a domain engineer could create (part of) the specification of Figure 2 using the method invocations in Figure 3b. Although this may be a bit clumsy, to make data entry easier, a form-based interface could be designed for little cost.

Notice that there is an inheritance hierarchy in the Java classes that would mirror the abstract syntax in a funny, inverted way. (The abstract syntax for the subclass would normally have a field whose instance was an instance of the superclass.) Notice that the definition of element is quite complex, including methods for adding and removing links in addition to an initialization procedure, written "*element(***String** *id,* **Integer** *own,* **String** *corp, . . .)*" where the ellipsis would include all the instance variables of the class. One would write such an initializer for each of the classes — e.g. *CommunicationsSatellite* and *Protocol* — (not done here, for brevity) and then invoke them as in Figure 3b. Also, notice that in order to link the elements together into a graph structure, one has to have an *elements* structure holding all the elements and a facility to look them up, *elementNamed*.[8]

Some interesting implementation decisions have been made when building this set of class definitions. First, instead of defining a class for each of the set-type fields, such as protocols and sequence elements, the implementation uses a "Hashset" collection device, and then relies on

[7] One would generally not make the instance variables public — used here for brevity — but would rather use constructors for each class and public accessor functions to expose the parts. The examples throughout the paper are intended to be "indicative" by nature. I am not a true expert in any of the languages or support tools! The examples are incomplete, but every attempt has been made to keep them mutually consistent.

[8] Not defined in Figure 3a.

```
public class Element
{
    public String uniqueID;
    public Integer owner;
       // 1 => Australia; 2 => EU; 3 => USA
    public String ownerCorp;
    public Position pos;
    public Hashset inProtocols; // of protocol
    public Hashset outProtocols; // of protocol
    public Hashset inLinks; // of linkage
    public Hashset outLinks; // of linkage
    Element(String id, Integer own, String corp) // ...
    { uniqueID = id;
     owner = own;
     ownerCorp = corp;
     //...
    };
    public void addInLink(Element iEl)
       {inLinks.add(iEl);}
    public void addOutLink(Element oEl)
       {outLinks.add(oEl);}
    public void delInLink(Element iEl)
       Iterator it;
    Object recent;
       {it = inLinks.iterator();
       while (it.hasNext()) {
               recent = ((Element)it.next());
          if (recent == iEl)
               it.remove();}
    }
    public void delOutLink(Element oEl)
       { // see above
    }
}
public class Satellite extends Element
{
    public Time initial;
    public Velocity v0;
    public Hashset sequencing; // of SequencingRule
}
public class SurveillanceSatellite extends Satellite
{
}
public class CommunicationsSatellite extends Satellite
{
    public Integer multiplex;
    public Integer rate; // in MBS
}
```

Fig. 3a. Java Domain Model (1).

```
public class SurveillanceController extends Element
{
    public Time openFrom;
    public Time openTo;
    // openFrom must be less than openTo
}
public class GroundStation extends Element
{
    public Hashset upProtocols;
    public Hashset downProtocols;
    public Hashset upLinks;
    public Hashset downLinks;
    public Angle FieldOfView;
    public void addUpLink(Element fEl)
       {upLinks.add(fEl);}
    public void addDownLink(Element dEl)
       {downLinks.add(dEl);}
    public void delUpLink(Element fEl)
       { // see delInLink in element definition
    }
    public void delDownLink(Element dEl)
       { // see delInLink in element definition
    }
}
public class Protocol
{
    public Integer security; // 1 => public, 2 => private,
                                      // 3 => secret
    public String pathType; // S1 or P8
}
public class Velocity
{
    public Time latitudinal;
    public Angle longitudinal;
    public Double altitudinal;
}
public class SequencingRule
{
    public State whenState;
    public Hashset actions;
}
```

Fig. 3a. Java Domain Model (2).

casting to make sure the elements are of the right type. Of course, this increases the potential for errors to creep in, but one might be able to isolate the programmer entirely from the decision, for example, if a form-based interface were always used to construct the virtual specifications. Some interesting little shortcuts in the concrete syntax show up even in this small example (Figure 3b). The *latitude* and *longitude* were converted to a decimal representation to input the position. The singleton protocol "*P8*" was introduced into each hashset to represent the "*Up*" and "*Down*" *protocol* links. Here the *country* and *privacy* descriptions were converted to the appropriate integers — normally represented as enumerated type elements in languages that have them. Although one might use final public static variables for this, an important

```
inP = new Hashset;
outP = new Hashset;
proto = Protocol(2,"p8");
inP.add(proto);
outP.add(proto);
el = CommunicationsSatellite("ATTs0088", 3, "ATT", Position(21.3, -157.867), inP, outP, 4096, 100);
elements.add(el);
*   *   *
el.addOutLink (elements.elementNamed("Alice1"));
el.addInLink(elements.elementNamed("Alice1"));
```

Fig. 3b. Java Satellite Program Fragment.

point here is that there is wide latitude available for mapping into an OO representation.

Haskell as a DSL. A language with which this approach to DSL design has been particularly successful is Haskell [Elliot99, Hudak00]. The approach is simply to design a set of data structures and functions whose use with one another characterizes the execution structures of the language. These functions are organized into a so-called "combinator library" for use by the application engineers. Haskell is especially useful when a "scripting language" is an appropriate model for domain activities; such languages are normally associated with simple state machine commands, such as with operating system shell scripts, but they are quite frequently useful for describing simple processes in particular domains.

One could consider using such an approach for the behavioral part of the specification in Figure 2. The structural part must be characterized as well, but it will primarily just be a syntactic variant of the Java declarations above.[9] An important aspect of analyzing dynamic aspects of the Satellite domain, such as which ground stations can talk to which satellites at any given time, will be to determine the exact position of the satellites. This is essentially a straightforward (though very complex, in reality) physics calculation, which can be expressed as readily in Haskell as in any other language. In the early 1990s, Haskell was used to describe coverage by AEGIS ships of enemy target regions [Carlson93]. A rather elegant conceptualization allowed them to pass functions where most languages would require data structures to be passed. The idea is simple: think of a Region as a mapping between a Point and a Boolean, whose value, when applied to any particular point, tells whether the point is in the region or not. One expresses that the Region is a mapping from Points to Booleans in Haskell as:

```
Region :: Point -> Boolean
```

They defined combinators to represent various common region shapes, such as circle of radius R, centered about the point.

```
circle :: Radius -> Region
```

Notice that this function, that returns a function as its value, is easily expressed in some languages, but its correct usage is fully type-checked in Haskell. In the satellite domain, we could use this same notion to describe the dynamic coverage of a satellite as the intersection of the satellite's broadcast cone with the earth; call this function its *coverage*.

```
Coverage :: Satellite -> Region
```

One could then determine which ground stations can listen to the satellite by simply enumerating the set of ground stations and keeping only those where the satellite's region of coverage included the ground station's location, viz.

```
[(sat,gs) | gs<-groundstations,
    sat<-satellites,
    (coverage sat)(location gs)]
```

One reads this "list comprehension" as: the set of all satellite — ground station pairs, (sat, gs), for which the region "coverage sat" applied to the point "location of gs" is true, i.e. the ground station is within the circle covered by the satellite's transmitter. Imagining a sphere centered about the earth with the satellite's distance from

[9] Actually, because of the lack of inheritance in Haskell's algebraic data types, the structure will rather resemble the flattened-out instance structure described below for Access in the section on COTS-based approaches or will be encoded in datatype extensions and overloaded classes.

the center of the earth as radius, the same mechanism could be used to determine which satellites can hear which ground stations' transmitters.

These are all the technical calculations of coverage. The point here is that they can be expressed elegantly, but they will probably not be used by the application engineer. Rather, a language fragment characterizing the sequencing information given in Figure 2 might be required. Briefly, the intent of the sequencing information is to provide enough of a model of the activities on-board a satellite to know when there will be enough power for operating transmitters and receivers, as well as predicting the necessity for such activities. The model allows the specifier to introduce arbitrary "instruments," such as "camera," and characterize them with simple finite state machines. These machines can transition from one state to another based on specific external stimuli — such as the solar cells being in sunlight or darkness, and signals being received or transmitted — and on transitions of other instruments on board the satellite between states (in the previous transition) — e.g. the receiver entered the "data ready" state.

To express the collection of state machines in Haskell, one might write the Haskell program in Figure 4b. To understand it, first consider Figure 4a. Often, one extends Haskell to become a DSL by constructing what is called a "monad," a datatype that allows one to hide state information while still affording an applicative language interface. Constructing these is nontrivial and requires at least as much — but different! — expertise as using any other syntax-based support mechanism. The type *Simulation* in Figure 4a is such a type; it involves a *Snapshot* of the status of all of the modeled elements, along with the stimuli added since the last snapshot and the time the snapshot was taken. It is not important here to understand why a mapping between *Snapshot*s is needed (nor what the extraneous type variable a is used for), just that *Snapshot* must capture all of the varying state information required to perform a simulation.

So the type *Snapshot* comprises a list of *Status* tuples along with a list of *Stimulus* descriptions and a universal time. Each *Status* tuple comprises a *string* — the element name -, *position, velocity*, and a list of all the *instruments* associated with the element, together with the *State* they are in as of the last "tick." The *tick* function is our way of causing a *simulation* to progress. Given a time interval since the last Snapshot was taken, tick must update the Status of all the elements, it must compute a new set of stimuli based on

```
type Simulation a = Snapshot -> (Snapshot, a) - the simulation "monad"
type Snapshot = ([Status], [Stimulus], UniversalTime)
tick :: Time -> Simulation ()
tick deltaT = \ sn @(e,ut). ((map e updateStatus, ut+deltaT), ())
                  where updateStatus (n, p,v,is) = (newp p v ut deltaT, newv p v ut deltaT, map is nexts)
                         nexts = . . . - must incorporate new stimuli and determine what new triggers arose . . .
observe :: Simulation Snapshot - like getChar
command :: String -> Simulation () – like putChar
type Status = (String, Position, Velocity, [(Instrument, State)])
type Rule = (State , [Reaction])
type Reaction = (Stimulus, State)
data Stimulus = Dark | Light | Receiving | Transmitting | Otherwise | Change Instrument State
data Instrument = Inst {name: String, current: State, rules: [Rule]}
data Owners = Australia | EU | USA
data Element = GroundStation {uniqueID, ownerCorp:String, owner: Owners, . . . , stat:Status, rules: [Rule] }|
                CommunicationSatellite {uniqueID, ownerCorp:String, owner: Owners, . . . , stat:Status, rules:
                                             [Rule] }|
. . .
next :: Instrument -> [Stimulus] -> State
next inst = – Find appropriate rule and test for reactions
type State = String
type Instrument = String
ut0 = 333252323 – zero universal time
comsatA = 110 – communications satellite standard altitude
```

Fig. 4a. Haskell Simulation Specification Fragment.

```
elements = [(CommunicationSatellite "ATTs0088" USA "ATT" ...
            ((Position 21.3 -157.867 comsatA), (Velocity 0 0 0), [("transmitter", "off"); ("receiver", "on")])
            [("transmitter", [( "on" , [(Light , "off")]; ("off", [((Change "receiver" "data ready" ) ,"on")])])
             ("receiver", [("on", [("data end", "data ready")]; ("data ready", [(Otherwise, "off")]; . . .])]))
           (SurveillanceSatellite "AF1102" ...
            [("camera" [("on", [(Dark, "off")] ...)]) ...]
```

Fig. 4b. Haskell Satellite Specification Fragment.

which instruments changed state, and it must add the time delta to the universal time. Again, the function definition here is mostly suggestive and need not be understood. The roles of analyses and input to the simulation are captured in the *observe* and *command* functions, respectively. They respectively map information out of the Simulation and insert stimuli into it.

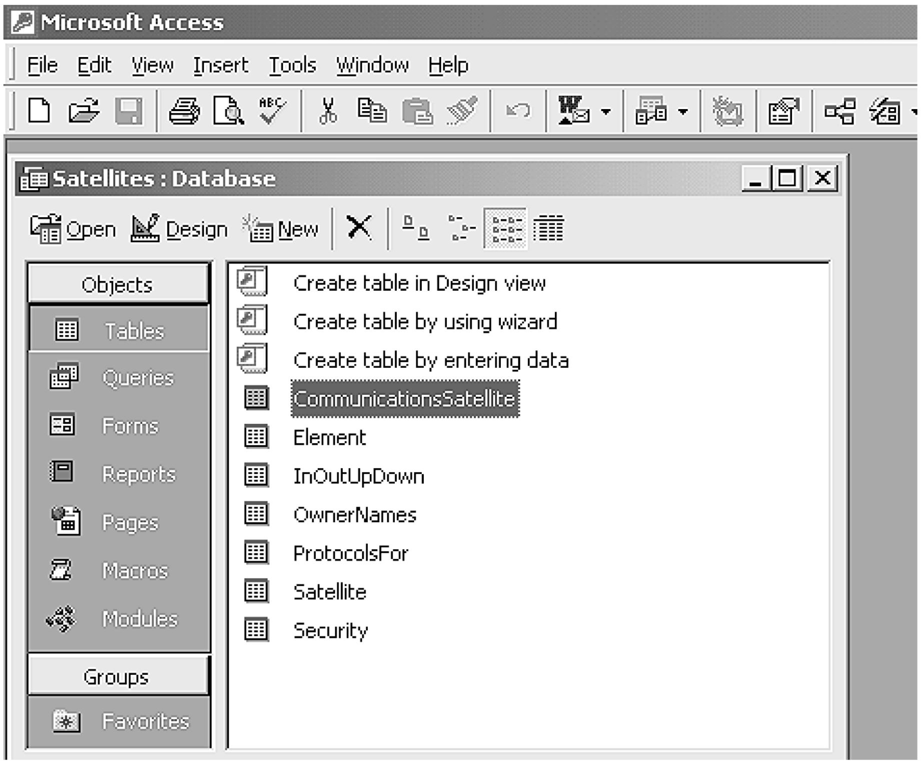
The rest of the program in Figure 4b is straightforward, using Haskell's data and type statements to define the structures somewhat similarly to the Java specification in Figure 3a. Notice that the abstract datatype construct, *data*, allows one to enumerate values easily, as in *Owners*. Notice that *Stimulus* is a similar list with the additional possibility of adding a *Change* record describing which *Instrument* entered which *State*. Probably the most interesting thing is that a domain engineer need not really understand what technically are the lists of tuples that are being entered in the elements declaration, but simply needs to understand the form used to express them. Since the specification in Figure 4b is all that needs to be written by the domain expert, very little of the complexity (or elegance) of Haskell for expressing this language comes through. That is, for representing the domain, this language extension is not too bad.

Of recent interest in the research community is the use of so-called "generic programming" [Jeuring95, Lopez97] techniques, wherein functions have arguments that are programs, types, type constructors, class hierarchies, or even grammars. It is possible that these can be used to provide higher-level abstractions whose mappings into the language effect DSL-like constructs. This approach can provide more extensibility than one would get from simply staying within the language. For example, although the expression above is not too far from the specification language in Figure 2, using generic programming one might be able to make it *exactly* the same. The generic program's evaluation would simply produce the description in Figure 4b from the description in Figure 2 as an intermediate result in the computation of the Simulation. As we shall see, this has all the benefits and pitfalls that a LEX/YACC approach has, when no intermediate representation is generated and maintained.

3.3. COTS-based Approaches

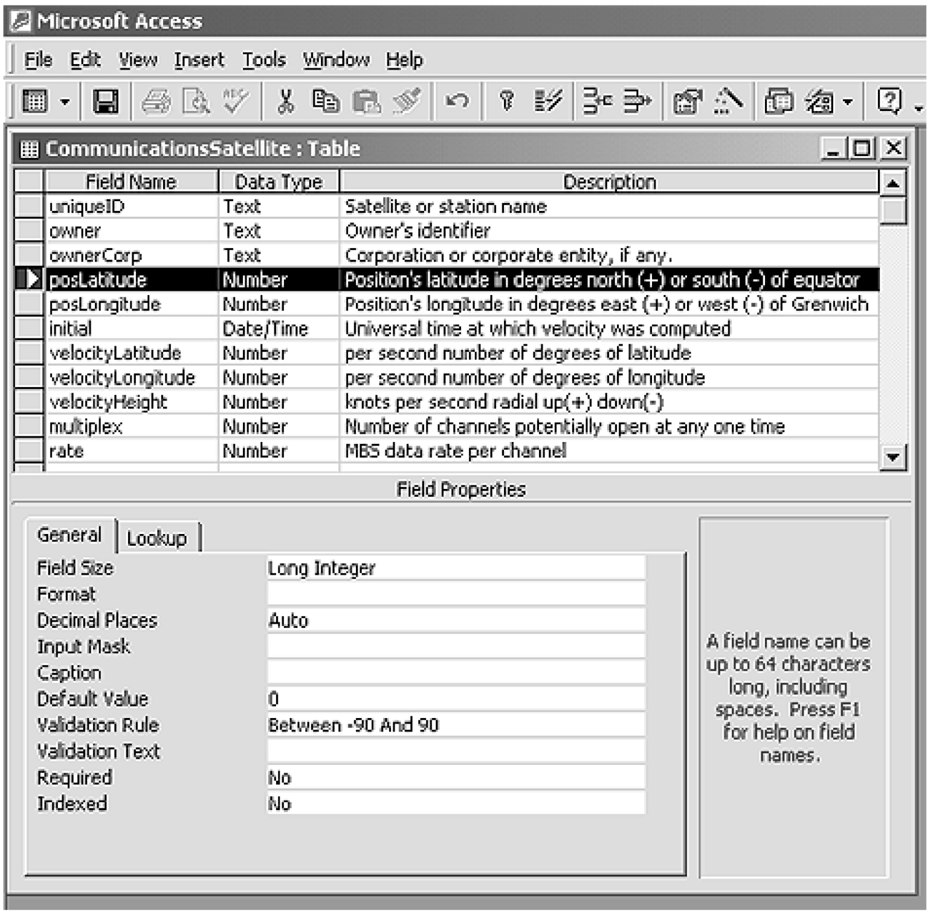
Another approach comprising several alternatives, that has gained some popularity is to express the structural aspects of a DSL in terms of Common Off-The-Shelf (COTS) products, such as Microsoft Access or, more commonly, spreadsheet representations.

Microsoft Access as DSL. Figure 5a illustrates some of the major structures of the satellite domain expressed in Access database schemas [Microsoft Access], where types correspond with "Tables"; Figure 5b illustrates the *Communications Satellite* type of Figure 2 from the relation schema *Design* view of Access. It is worth comparing this definition with the corresponding definitions from Java, for example. Here, the entire supertype structure has been unfolded into the lowest level type. In particular, the attributes from *Elements* and from *Satellites* are all present in the *CommunicationsSatellite* "Table." Even though this is not necessarily how one would design a schema for this DSL in Access, it certainly represents a viable option. The alternative is to reference *Satellite* fields, and then *Satellite* could reference the *Element* fields, using *keys* of the relations. The present definition makes data entry easier. Fortunately, Access has a quite intuitive interface for creating instances and thus is not much more cumbersome than conventional languages to use. Notice that there are no fields for the *Protocols*. This will be discussed shortly.



Create table in Design view
Create table by using wizard
Create table by entering data
CommunicationsSatellite
Element
InOutUpDown
OwnerNames
ProtocolsFor
Satellite
Security

Fig. 5a. Access Tables for Elements and Enumerated Types.

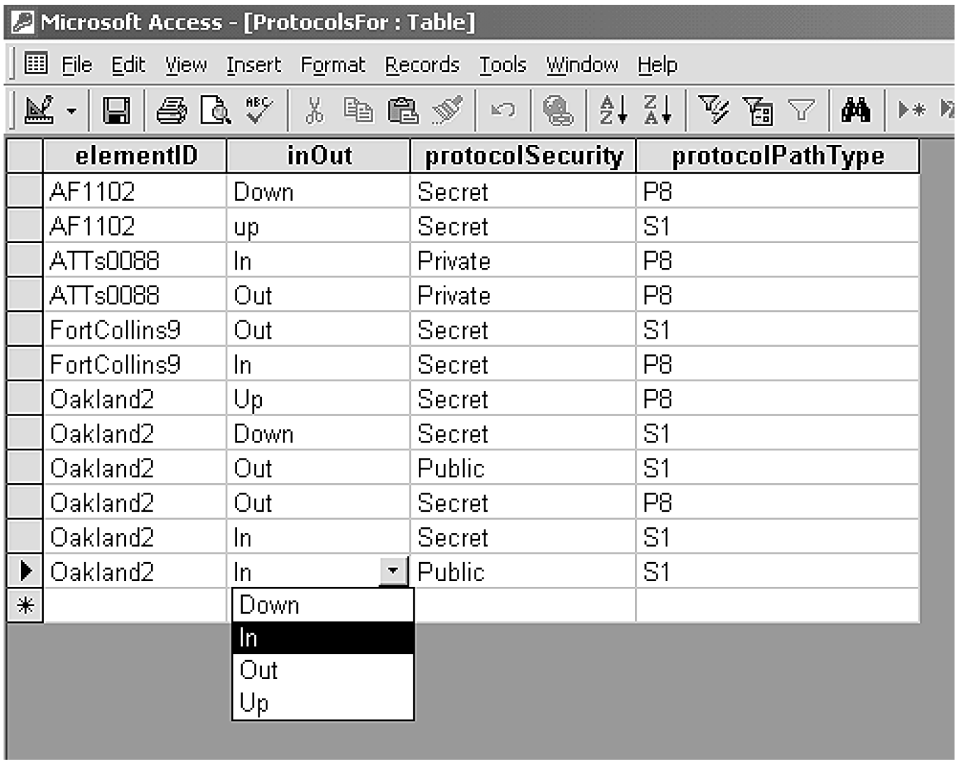

Microsoft Access
File Edit View Insert Tools Window Help
CommunicationsSatellite : Table
Field Properties
General Lookup

Field Name	Data Type	Description
uniqueID	Text	Satellite or station name
owner	Text	Owner's identifier
ownerCorp	Text	Corporation or corporate entity, if any.
posLatitude	Number	Position's latitude in degrees north (+) or south (-) of equator
posLongitude	Number	Position's longitude in degrees east (+) or west (-) of Grenwich
initial	Date/Time	Universal time at which velocity was computed
velocityLatitude	Number	per second number of degrees of latitude
velocityLongitude	Number	per second number of degrees of longitude
velocityHeight	Number	knots per second radial up(+) down(-)
multiplex	Number	Number of channels potentially open at any one time
rate	Number	MBS data rate per channel

Field Size	Long Integer
Format	
Decimal Places	Auto
Input Mask	
Caption	
Default Value	0
Validation Rule	Between -90 And 90
Validation Text	
Required	No
Indexed	No

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

Fig. 5b. Communications Satellite Table Fields.


Microsoft Access - [ProtocolsFor : Table]
File Edit View Insert Format Records Tools Window Help

elementID	inOut	protocolSecurity	protocolPathType
AF1102	Down	Secret	P8
AF1102	up	Secret	S1
ATTs0088	In	Private	P8
ATTs0088	Out	Private	P8
FortCollins9	Out	Secret	S1
FortCollins9	In	Secret	P8
Oakland2	Up	Secret	P8
Oakland2	Down	Secret	S1
Oakland2	Out	Public	S1
Oakland2	Out	Secret	P8
Oakland2	In	Secret	S1
Oakland2	In	Public	S1

Down
In
Out
Up

Fig. 5c. Protocol Table.

Some other shortcuts have been used in designing this representation. For example, what was a *Position* type in Java, with latitude and longitude fields, has been unfolded into the definition of the satellite. Notice also that there is some flexibility in using SQL, the well-known database query language, to somewhat constrain the field values. In Figure 5b, we are focused on the *posLatitude* field. Notice at the bottom of the screen image that there is a "Validation Rule" restricting the values in this slot to be between -90 and 90 (degrees). ("Between x And y" is a formal SQL constraint.)

Figure 5c shows how the protocols could be dealt with. When designing relational schemata there are always two choices that can be made. A single relation with several fields can be used to characterize objects that form "keys" of the relations, that is, a value that is present exactly once in any given row. The other way to organize relations is the so-called "entity-relationship" style, where simple binary relations relate objects to property values. The relation name is the property. This is better for representing multiple-valued attributes, such as the *protocols* and *links* in this example. The *elementIDs* do not form keys in this kind of relationship.

The schema for the relation used in Figure 5c is an adaptation of the entity-relationship style in which the last three columns really form a single "value" expanded in a fashion similar to that done to *Positions* above. Figure 5c itself shows how one would enter the *data* itself, rather than the schema definition. Another feature of the database metaphor is that a table of constants — *up, down, in*, and *out* — has been defined and the values of the column *InOut* have been restricted to these values. The cursor is poised over the *InOut* column for the last entry, whence the drop-down list of choices is being offered to the user entering the data.

XML as DSL. Another example of COTS-based tool approaches to specifying DSLs are the purely abstract-syntax-based XML approaches that have been becoming popular recently, along with, for example, DOM-based tools for mapping over the abstract-syntax-captured in the XML representation [XMLSoftware]. These represent the minimal interface and syntax approach, leaving the meanings of things entirely to the interpretation of the tools they are used with. There is also a variety of structuring mechanisms, that take on the role of grammars or abstract syntax specifications, such as RDF

```
<CommunicationsSatellite id= "satellite. ATTs0088.007.XX3"
                         name = "ATTs0088"
                         owner = "USA"
                         multiplex = "4096"
                         rate = "100 MBS">
    <ownerCorp> ATT </ownerCorp>
    <position latitude = "21.3" longitude = "-157.867" />
    <inProtocol> Private P8 </inProtocol>
    <outProtocol> Private P8 </outProtocol>
    <outLink to = "groundstation.Oakland2.007.XX7" />
    <inLink from = "groundstation.Oakland2.007.XX7" />
</CommunicationsSatellite>
```

Fig. 6. Possible XML version of the ATTs0088.

and XML- schema. One has a difficult problem mapping to these representations if there is no more flexible interface available than text editing — i.e. they make horrible languages for humans to write — but this approach is very valuable as an *adjunct* approach to other approaches. As one can see from Figure 6, an XML expression of just the ATT satellite above is quite a bit more verbose than even the Java version, and considerably more so than the true DSL specification in Figure 2.

Microsoft PowerPoint as DSL. We at Teknowledge are pursuing a hybrid approach, between COTS-based user interfaces (in particular PowerPoint [Microsoft PowerPoint]) and our traditional syntax-based approach developed in the 1980s and 1990s using program transformations for analysis and translation in the Popart system [Wile86,Wile93]. Within our extension to PowerPoint, called the Design Editor [Goldman99], the application designer designs a graphical interface comprising icons and different kinds of arrows. The designer also declares what attributes may be used to decorate the various classes and superclasses of component and connector (icon and arrow) types. Domain types can also be enumerated. This, so-called "architecture style"[10] or "domain design" takes the

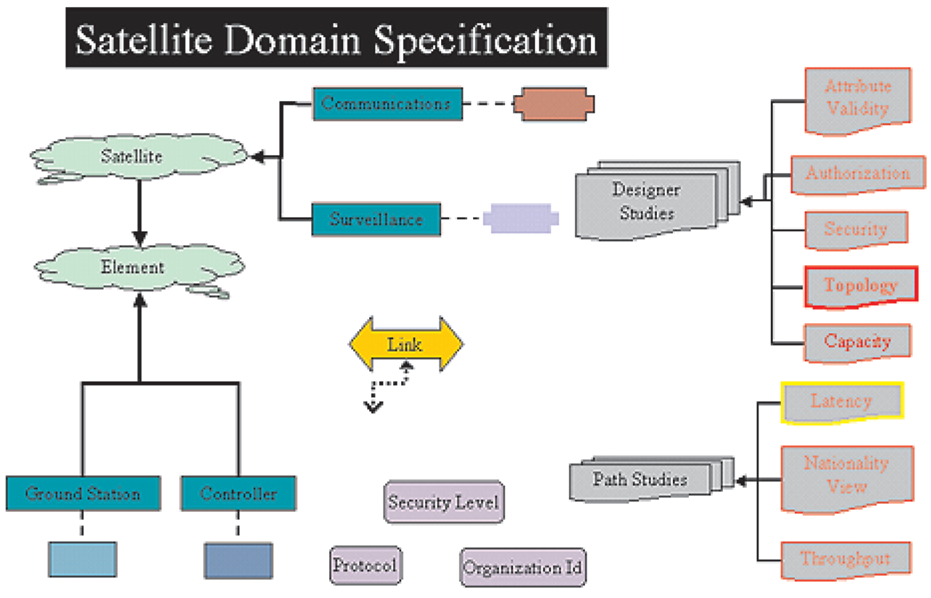


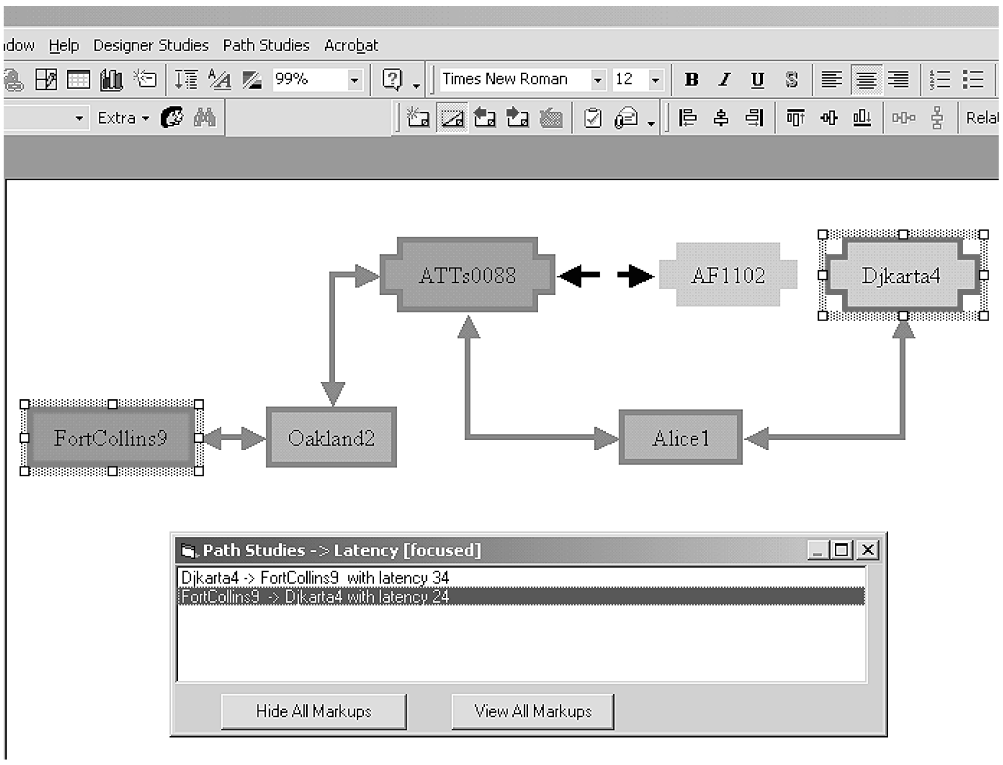
Fig. 7a. Satellite Domain Specification in PowerPoint Design Editor.

[10] We originally designed this to provide specific notations and analyses for software architectural styles inAcme [Garlan00], but later recognized its potential for designing structural portions of DSLs.

place of a syntax for the structural aspects of the domain. Specifications can then mix graphics with textual and graphical attributes in this hybrid DSL technology. Figure 7a illustrates the style specification for the satellite domain; Figure 1 is an instance of a design expressed in these domain terms. Various icons here represent classes of the "style" domain. The "clouds" are abstract classes with which the domain designer can associate attributes; here, all the slot variables of the Java specification are attributes. The mechanism for specifying these attributes is form-based and rather "clunky," but one can express that an attribute can be optional, required and / or multiple valued. Its type can be proscribed and ranges given for numeric types.

There are four different kinds of components, linked to the clouds via subtype arrows. Each has an associated icon (in different colors) — cruciform shapes for satellites and rectangles for controllers. There is only one kind of connector, called a "*Link*," represented by a dotted, double-arrow. The three rounded rectangles labeled *Security Level, Protocol*, and *Organization ID* are enumerated types for this domain. Right clicking on them brings up a menu to enter the constants. The two icons labeled *Designer Studies* and *Path Studies* are "analysis groups;" they will become major menu labels at the top of the PowerPoint screen when editing designs in the Satellite style. The menus will contain various analyses as selections, *Attribute Validity*, *Authorization*, etc. With each analysis one can associate parameters that must be filled in when the analysis is invoked during satellite system design and analysis time.

Figure 7b shows the original design being analyzed for *Throughput*. Notice that the PowerPoint frame segment shows the editing tools specialized for this domain in the upper left-hand corner just above the editing window. The *Djkarta1* satellite and the *FortCollins9* ground station have been selected, so after mousing the *Path Studies > Throughput* analysis menu in the menu bar at the top of the frame, the analysis computed throughput values in both up and down directions between these two elements.[11] Recall that the inclusion of this menu was indicated by the presence of the analysis hierarchy on the domain design in Figure 7a. When we se-



Djkarta4 -> FortCollins9 with latency 34
FortCollins9 -> Djkarta4 with latency 24

Fig. 7b. Throughput Analysis.

[11] There is not enough information in our model to do this. The example illustrated is from a style that is slightly more complex in some dimensions than the one discussed here.

lected the second analysis result in the analysis window (at the bottom of the screen) the path with the maximum throughput (here, the only path) was highlighted on the original diagram, i.e. the feedback from analyzers is presented right on the original diagram whenever possible.

Our approach is part of a broader category of approaches that also includes techniques involving the use of UML [Booch99] with, for example, COM-based domain-specific analyzers.

4. DSL Support Issues

Each of the techniques presented in the previous section has advantages and drawbacks that will be detailed further in the sequel, but first it is important to recapitulate why providing DSL support is potentially so difficult. Generally, there are three kinds of issues that make DSL support difficult: issues specific to the fact that we are designing a DSL, more generic issues that have already arisen in providing support for general-purpose programming languages, and pragmatic issues that transcend technical arguments for particular solutions. Before discussing in further detail how each of the DSL support approaches above impacts these issues, a slightly more careful discussion of each issue is presented in the following sections. In the following section, Section 5 — Support Advantages for DSL Approaches, a table is presented that lists various issues as row headings and various support techniques as column headings. The **boldface** labels in the sections below will be used as the issue labels in that table.

4.1. DSL-Specific Issues

Above we focused on the design and specification of the syntax of the DSL and abstract or internal representations, demonstrating in particular, that the appropriateness of the specification language is very domain-specific, sometimes even "sub-domain-specific." Some aspects are easily characterized with traditional BNF-like syntax, others yield better to tabular approaches, and still others, to graphical representations. Worse, most domains have some aspects of well-known languages already present, e.g. infix operator notation, with only a limited amount of truly new syntax added for the problem domain. Some ideally require a mixture of graphical, syntactic and even tabular entry.

It will be useful below to separate the language design issues into four separate concerns: specifying the **structure** of objects or concepts of the problem domain, specifying **relationships** between objects or concepts in the domain, specifying (other) **constraints** on objects or among concepts of the domain, and specifying **behavior** of domain objects or agents. Some DSLs have little or none of one or even two of these issues; they are all almost always present, but may be implicitly dealt with by the infrastructure. For example, one need not specify in the satellite domain the constraints that actually yield computations of total path bandwidth; the analytical infrastructure provided by the DSL support designer will provide the algorithms.

Specific problems that arise include:

- Sub-languages present in a DSL are "reinvented" rather than adopted directly, viz. the languages for constraints, for process, and for the various constants. It would often be better to adapt or adopt well-known languages for such purposes, especially if there are no precedents for expressing these in the problem domain jargon.
- The language itself may be a bit artificial for the domain. In particular, the syntax used in the example in Figure 2 for how to connect the different elements is clumsier than simply drawing the connections as was done in the motivating Figure 1.
- Designing analyzers and simulators is made more or less easy by how well the implementation technology matches the natural model underlying computations in the domain. For example, if the underlying model is the simultaneous solution of a set of differential equations, as with many control system applications, translation into an implementation mechanism like MatLab may be the best approach. Here, for example, the Haskell simulator technology would be quite appropriate for designing the tracking analyzers.
- Debugging simulations and analysis is made difficult because problems and errors have to

be related back to the problem domain, rather than in terms from the evaluation mechanism itself.

4.2. General-purpose Language Support Issues

All of the concerns that are germane to general-purpose programming language (GPPL) design and support may become important in the support of a specific DSL. The tool support necessary can often be classified in GPPL support terms: **editing**, dereferencing, saving; compilation concerns, such as parsing, **type checking**, and code generation; static **analyzer** designs, such as usage graphs; means for debugging DSL "programs," often entailing an "**unparsing**" activity to map back out of the internal representation; support for **persistence** and **versions**; and mechanisms for **simulating** or executing DSL programs. These issues will be elaborated in more detail below.

4.3. Pragmatic Support Issues

A final category of problematic issue, often critical in the decision of what approach to take to provide DSL support, is purely pragmatic: sometimes a particular implementation platform is important or sometimes the ability to spread effort among members of a team is critical. These constraints often cause technically surprising choices to be made.

- There may be sources of data and analyses independent from those with which the technology must interact. This is somewhat related to issues that arise in Computer-Supported Cooperative Work (CSCW) [Ben-Shaul95] where support of multiple sources for changes to the specification is required.
- Depending on applications, several engineers may need to cooperate to come up with a specification. The technologies for computer-supported cooperative work (CSCW) — transaction control and version management — are not incorporated, for example, in the Cornell Program Synthesizer Generator. This may argue for a much more fragmented approach than might seem ideal, possibly requiring different support options for different portions of larger DSL.
- The technology must run on the platforms the application engineers are familiar with. One almost never has the luxury of defining an entire programming and hardware support environment for the accomplishment of particular tasks. Interoperation with particular file formats, efficiency considerations, and operating system concerns can each dominate a technology choice decision.

Many of these problems will pertain to any of the alternative approaches to providing DSL support mentioned above. However, each approach will tend to solve or ameliorate at least one of them. It is worth revisiting each approach above to see whether problems are aided, exacerbated or untouched by the approach.

5. Support Advantages for DSL Approaches

Table 1 is an attempt to indicate how effective various approaches to providing DSL support are. The issues listed in the left column are keyed to the discussion above. *Structure, Constraints*, and *Behavior* all refer to how well different language aspects can be *expressed* in DSLs supported by the technology to the left, not how readily the semantic support behind these sublanguages can be expressed. Other rows refer to how difficult it is to provide support for the labeled activity, not whether it is already provided by the technology. The "$\Uparrow$" symbol indicates that outstanding support is provided by the technology; the "$\wedge$" means excellent; "$-$,"good; "$\vee$," fair; and "$\Downarrow$," poor or difficult to provide.[12] The table is meant to be purely indicative and certainly represents just my opinion based on my own experience with some of them. There is a wide variety of other specific support tools and several of the issues above have not been dealt with because of my lack of personal experience.

A word on why the columns were chosen is plausible. The choices simply represent a group of tools with which I have some limited familiarity. This has some major implications with

[12] Think of the scale in terms of "thumbs up" through "thumbs down."

	YACC	Syn. Gen.	Haskell	Java	Access	PowerPt. DE	XML	COM	
Structure	⇑	⇑	—	∧	⇑	∧	∨	∨	
Relationships	—	—	—	—	—	⇑	∨	—	DSL-
Constraints	⇑	⇑	∧	—	—	∨	∨	∨	Specific
Behavior	⇑	⇑	∧	—	∨	∨	∨	∨	
Edit	—	⇑	—	∧	∧	⇑	∨	∨	
TypeCheck	∨	∧	∧	—	—	—	∨	∨	
Analyze	∨	∧	⇓	⇓	—	—	∨	∨	GPL
Simulate	∨	∨	⇑	—	∨	∨	∨	∨	Support
Unparse	⇓	—	⇓	⇓	—	∧	—	—	
Persistence	∨	—	∨	—	—	—	—	∨	Pragmatic
Versions	∨	⇓	∨	∨	⇓	⇓	∨	⇓	Support
	Language Design		Language Extension		COTS-based Approaches				

Table 1. Effectiveness of Approaches
⇑ — Outstanding, ∧ — Excellent, — — Average, ∨ — Fair, ⇓ — Poor.

how the table should be considered. Notice in particular that the "PowerPoint Design Editor" labels one column. It is important to realize that were the comparison simply with PowerPoint itself, a much more bleak support picture might be painted, something akin to those of XML or COM. By the same token, were I familiar with SmartTools [Attalli01], that effects an automatic conversion from DTD (Document Type Definition) to abstract syntax trees and vice versa is supported, I might have rated its combination with XML comparable to the advantages from the Synthesizer Generator.

Other examples abound. Excel would make an excellent column label as well, but I have not used it in this context at all. Moreover, there are modern research versions of language processing tools, such as ASF+SDF [Heering89, van den Brand01] and LISA [Mernik00a] that may actually be superior to the Synthesizer Generator, if pragmatically acceptable to an organization. For a recent overview of such systems, see [Heering00].

In the following, each row of the table will be discussed in turn, since the table is most "indicative" of what to expect when using the various techniques.

5.1. Describing Structure

Explicit syntax is ideal for expressing complex declarative structures, such as in the *element* specifications above, where the delineation of attributes such as *location, type, name, position, velocity*, and *sequencing* are sometimes specified. Since explicit syntax is easily provided for in YACC and the Synthesizer Generator, they do an outstanding job of representing structure. Syntactic representations are especially useful when several optional parts of the structure may be present or where there are complex constraints between the attributes. Continuing the example, the *hours, field of view, multiplex, rate*, and *sequencing* are all optional and dependent on the element's type; syntax can easily provide for the facile expression of that fact. Syntax can also provide for correlations between attributes, such as the required presence of *state* declarations when states are used in *sequencing* instructions. Lists of items are also handled syntactically, especially when ordering is implied, but these are sometime handled better as relationships (see next item).

Programming language extensions are somewhat awkward in this regard, but they are perfectly usable when other features can be used effectively. Moreover, it is not difficult to describe form-based interfaces to these in general. Java's inheritance mechanism renders it a bit more concisely for expressing these fixed structures than Haskell, so the former was rated excellent and the latter merely good. Our form-based PowerPoint Design Editor is about the same as Java to use for specifying attributes of objects, assuming Java forms are created for attribute entry for each object type.

When attributes are very regular, a tabular presentation is often useful, so Access can shine through here and is also given an outstanding rating. Access has built-in forms for data entry and allows further customization of these by the designer. Access is quite awkward for interrelating both simple attributes and lists of attributes simultaneously, as was seen in the example above - Figure 5b, but recasting the lists of elements as relationships unto themselves — Figure 5c — redeems Access in this respect as well. Similar comments could also be made for COTS spreadsheet programs, useful when special simulations or analyses are easily expressed as relaxations.

XML and COM fare rather poorly here in that even though they can be used to express the same information as the others, the interfaces are quite verbose to say the same thing. They can be boosted easily enough with a good form-based interface, so they should not be discounted out-of-hand.

5.2. Describing Relationships

Pictorial representations seem to be preferred to syntactic ones when detailed relationships between objects are described. Hence, our PowerPoint Design Editor is rated outstanding here. PowerPoint itself would not be, for there the logical structure is intertwined with the physical layout structure so implicitly that there is considerable noise within the internal representation. The preference for graphical support pertains even with medium-sized specifications, where the direct interactions can be seen as arrows or lines between the involved components. Larger containment structures can be represented by implicit links between "slides" in PowerPoint as well, where one slide is the definition of an icon on another. Beyond that, PowerPoint joins the rest of the techniques in requiring some sort of file management to describe aggregates.

Syntactic support for describing relationships usually requires that objects be named uniquely or relatively to some enclosing structure. Generally, application engineers must use awkward techniques for finding what objects are referred to, by looking up their definitions. The programming language approaches must use similar naming conventions. Although Haskell can be used to express cyclic relationships, occasionally it is a bit mind-numbing to understand how it works. On the other hand, Haskell can deal with lists of objects better than any of the other representations, so it is considered as "good" as the rest of the syntactic and language extension approaches. There may be explicit tool support provided for finding object definitions in Java or Access environments, so that can make relationships easier for application engineers to deal with.

XML is the worst of the bunch for expressing relationships, for one must reinvent some notion of unique identifier each time such relationships are desired. COM's references are straightforward, but accessible only through programmatic means; they can look approximately like Java object creation code.

5.3. Expressing Constraints

Constraints — with essentially complex structures — are often best expressed syntactically, and hence, tools providing syntactic support are ranked outstanding, as the best way one can provide domain-specific support. Of course, specific domains may contradict this generalization and make considerable headway with iconic constraint symbols.

Language extension can occasionally be just as good at expressing constraints — that occasion being when no specific jargon of the domain has already been developed for expressing constraints and reliance on a "standard way" of saying them is desirable. Haskell is especially nice in this regard, and, accordingly, has been rated excellent. Haskell's lists, logical connectives and list comprehensions (see example above) are very useful here. Java can do all right here as well, but it is usually considerably more awkward, especially since it uses unconventional syntax for conditionals and statements. Access allows some local constraint specifications using SQL; moreover, a programmatic interface is available, that allows designers to provide for SQL's use on a more global scale. Hence, these two are still good for representing constraints.

The PowerPoint Design Editor and the interchange languages, XML and COM, are all equally bad here — the designer really has to provide an ad hoc constraint definition facility. Of course, one of the other techniques could be used for this in a hybrid approach, so no particular method should be discounted a priori. The fact that the interchange mechanisms exist makes the potential for combining effects very promising.

5.4. Describing Behavior

Almost exact comments that apply to constraints hold for behavioral representations too. Since Access has no way of expressing behavior, it is rated merely fair in this column, but otherwise all the entries are the same for the behavior row in the table as for the constraints. In fact, some people prefer flowcharts to a structured syntax for expressing sequencing, so the PowerPoint Design Editor could be bumped up a notch for some uses. Researchers raised in the structured programming era tend to eschew such potential complexities.[13] Even though graphical idioms may turn up occasionally, people tend to use a syntactic way to describe procedures, so I have generally concluded that syntax is an outstanding means for specifying behavior.

It is worth mentioning that although Haskell is rated excellent for expressing behavior, it is not for the same reasons as for constraints, for a completely different dialect of the language will usually be useful for expressing behavior. The most common usage is for scripting, using the monad construct described briefly above.

5.5. Editing

This is the forte of the Synthesizer Generator in that it understands the structure of the syntax and can prevent entry of erroneous text in the first place; hence, it is rated outstanding in this category. At least one other editor, the ubiquitous emacs, has been provided with "modes" that approximate syntactic understanding of the more popular languages. Sometimes a DSL can be designed to similar standards (like C) and take advantage of the fonting and bracket-matching features of emacs. In fact, since it is open- source, an expert can probably design such a package for a new DSL rather easily. For this reason, LEX/YACC and Haskell are rated good in this aspect. A Java emacs mode is also available, but Java is ranked excellent because there are programming environments built specifically for it, which facilitate editing to a much greater extent [Sun Forte].

Access and PowerPoint both have nice editing interfaces; to me PowerPoint's feels much less clumsy than Access, so I rated the former outstanding with Access only excellent. Naturally, as with almost everything, the interchange languages require that you design your own. However, if the XML used matches an html application to some extent, specific form editors may be around to make editing it a somewhat less unpleasant chore than it is normally using a text editor!

5.6. Defining Type Checkers

This is another area where the Synthesizer Generator excels, in that providing a set of attribute grammar declarations to do type checking is generally rather straightforward, but tedious. The computation of the attribute has to be expressed for each construct in the language, but their expression is equational in terms of attributes of parent nodes (inherited) or children nodes (synthesized). The relaxation computation proceeds until it converges.

In fact, LEX/YACC has been used long enough so that this activity is not too difficult for experts there either, perhaps requiring frequent reference to "the Dragon Book" [Aho86]. This is the area where support tooling really requires unusual (read, "expensive") expertise, however,

[13] This raises an interesting issue: to what extent should the DSL designer imbue the language with good design principles? Generally, if there is already an established practice in the domain, we have found it best not to perturb it; dealing with the technology itself is often such a hurdle for application engineers that learning new notations can be the "straw that broke the camel's back."

and is probably the main drawback to providing syntax-based DSL support.

There are two issues related to using language extensions for DSL support in this category: (1) how much of the language's own type checking can be used to provide effective DSL type checking and (2) how hard is it to express what is left over? The table rankings are for (1), where Haskell's native type checker is nothing less than astounding — programs that type check often run correctly the first time! Java's strict adherence to class accessor syntax can also provide good support for type checking. Regarding (2) these languages can often be regarded as *poor* for the following reason: with the exception of the data structure declarations, for which type checking programs can be written, when the user writes an actual program in the language, there is no abstract syntax representation of the program that a type checker can refer to. So, for example, if only one of two specific functions should ever be called in a particular context, establishing that domain-specific type constraint might require the introduction of extraneous variables at run-time to signify the correct context and the number of times one of the functions had been called. These variables have nothing to do with functionality, and if the type constraint is critical, it may be too late to delay such constraint checking. This inability to separate concerns can be damning for a language extension approach. The next best thing is to write an *interpreter* in the language over an abstract syntactic representation so this kind of type checking can be data-driven at analysis time.

Access and the PowerPoint Design Editor are rated good simply because their interfaces are good at enforcing adherence to a rigid syntax. Otherwise they are only fair, possibly requiring extensive programming; XML and COM can require similar amounts of programming.

5.7. Describing Analyzers

Most analyses — for usage relations, inaccessible code regions, improper initializations, etc. — share the same technology and criticisms just enumerated for type checking, because in some sense, type checking is the quintessential *static* analysis activity. The rows differ only in that Haskell and Java representations share the problem mentioned above: their lack of an abstract syntactic representation of the program makes it difficult to perform analyses directly. With one caveat: if the sole purpose of the specification is to perform a particular analysis, the evaluation of the functions themselves may perform the analysis.

An example may help clarify this. Assume that we have a program extension for a musical dialect in Haskell comprised of functions C, C#, D, D#, E, F, F#, G, G#, A, A#, B (with some equivalent functions, such as Db). Each function takes two integers: one represents an octave and the other a tone duration. One could then construct a tune as a sequence of applications of these functions in a "MIDI monad," e.g.

C 3 4; C 3 4; E 3 4; E 3 4; F 3 4; F 3 4; E 3 4
. . .
(Twinkle, twinkle little star)

(Do not be concerned if the technical details seem obscure — the point is that a program could play the song you entered using this somewhat clumsy technique.)

If we want to analyze this program to see whether it contains any scale progressions (a particular pattern of notes), what can we do? All we have is a big composed function and no data representation for it. However, if *all* we wanted to do was analyze the program for scale patterns, we could define the functions themselves to check for them. But then imagine we want to check for repeated themes. We have the same problem. Of course, there may be more generic ways of handling this problem, perhaps making these functions of a class or perhaps even using generic programming, but the applications of program extension so far have not dealt with this problem — i.e., they have all been single-use extensions.

5.8. Simulation

Simulation is the quintessential *dynamic* analysis activity. As we demonstrated in the satellite example above, Haskell is regarded as an excellent language for simulation. It is outstanding in some contexts, but awkward in others, especially those requiring relaxation or search. Sometimes these can be programmed by relying heavily on the lazy evaluation mechanism or by building the search into a monad,[14] but other languages such as Prolog [Lämmel01] or simulation packages such as MatLab may be a better choice for some tasks. Java is another reasonable language for using the language extension as a direct program in the domain. Again, we are relying on the application engineer to write programs (as in Figure 4b) whose evaluation runs a simulation of the implied behavior.

All other approaches require programming the simulator in some language and provide no specific leverage on their own for this or other evaluation activities.

5.9. Unparsing

Sometimes one needs to report errors or results of analyses back in terms of the original domain. This unparsing activity is not difficult for approaches that keep an abstract syntax representation of the original program, such as Access and the Design Editor, and in fact, the XML and COM representations as well. The Design Editor, moreover, provides a facility for highlighting the original objects on the PowerPoint slides, so it has been ranked as excellent for supporting analysis result reporting.

It is more difficult for the syntactic approaches to provide unparsing facilities, especially the YACC approach that does not even require inventing an abstract syntax in the first place. The Synthesizer Generator does provide a linkage between abstract and concrete syntax, so it could be programmed there. Again, since the language extensions generally have no abstract syntax, reporting can be nearly impossible to phrase in domain terms with them. If the extensions are used extensively for representing data structures of the domain, the debugging facilities of the programming environment for the language may suffice to describe the problem structures. This is an especially serious problem with generic programming approaches.

5.10. Persistence

The Synthesizer Generator, Java programming environments, Access and PowerPoint all provide means for saving files, so they are at least good choices if the specifications in the DSL need to be kept persistent. (It is not a foregone conclusion that persistence is necessary; the purpose of the DSL may just be to express web search queries, for example, something of generally transient utility.) YACC and Haskell require extraneous mechanisms, and thus are only rated fair. XML is rated good because there are emergent facilities for keeping XML databases persistent [Dashofy01]. I know of no persistence mechanisms for general COM structures, so I presume such facilities must be programmed from scratch. This will be a nontrivial activity in general, since the structures may be cyclic and not refer to printable items, but on an ad hoc basis they should be manageable.

5.11. Versions

Versions is a placeholder here for all the scaling issues allowing large development efforts to use DSLs. This requirement is almost non-existent in the efforts to date, that have used a DSL-based approach. It is even possible to argue that it will never be a serious problem: large problem solutions are the result of many small problem solutions, each of which is best expressed in a DSL close to the individual problem space.

Even with this argument, there will be problems of intercommunication of shared data, version management, translation between representations, etc., not to mention coordination of analysis and simulation activities. The table row here is the one I understand least well, but I am inclined to think that the Synthesizer Generator, PowerPoint and Access are the least likely to become scalable on their own. COM is an in-memory representation and certainly is not intended for massive amounts of data either.

In fact, DSLs change more frequently than general-purpose languages, exacerbating versioning problems. But there is good reason to believe that language development facilities of the future will be able to cope with such rapid change, for example, based on LISA. Its tool support allows incremental development with multiple attribute grammar inheritance [Mernik00b].

[14] For example, the list monad gives all executions over all elements of a list.

Some recent efforts have at least examined the efficiency penalty incurred by using DSL approaches. Unlike extension approaches, language-based paradigms may actually be more efficient than the obvious program written in a general-purpose language, because knowledge of the domain can be taken into account in the design of the compiler optimizations Time will tell how important these pragmatic concerns become.

5.12. Hybrid Approaches

When one really must design a domain-specific language, LEX / YACC-based approaches seem to represent the low buy-in cost alternative. One forfeits any easy accessibility to editors or any other support facilities, but the existence of other support code written in C or C++ could easily make this an attractive choice for providing an interface to that functionality. LEX and YACC do not even require that an intermediate syntactic representation be used — an abstract syntax. However, its disciplined use to parse into, for example, an XML representation - playing the role of abstract syntax — could link the DSL specifications into more generally available tool support. This alternative definitely requires specialized expertise for DSL development. This approach I would characterize as a hybrid approach, because tools from two or more columns of Table 1 are used in constructing a DSL support environment.

Our research is best classified in this area. The PowerPoint Design Editor is presently the focal point of our support environment. However, notice the many areas in the table where it is deficient. In an application done for the Census Bureau [Wile00] we used PowerPoint to express questionnaire flow designs together with Access to express the database structure for question answers and together with a form composition program (Tarantula) to describe how to ask the questions in the different nodes of the flow design. Moreover, a syntax-based mechanism was used to parse an SQL extension to describe constraints on answers to questions presented in the forms. Idiosyncratic analyzers for Topology, Coherence, Branch Coverage, Producer/Consumer relationships, and other abstraction tasks were all written in Visual Basic, producing feedback on the original flow diagrams. We had planned to have a Haskell evaluation engine used to allow the designer to simulate taking surveys, but the funding was cancelled before it could be completed. Except for the XML-based Tarantula to PowerPoint link, these all communicated through COM representations.

The idea for our hybrid approach is basically to take whatever expressive power is needed to describe a task and find an appropriate mix of COTS components, specification templates and idioms in those components, and domain-specific language subdialects and then write as little code as possible to glue them together and provide appropriate analyzers and simulators.

It is easy to imagine UML as the center of another hybrid approach; we have one: a state chart simulation language using MatLab as evaluation mechanism linked to Rational Rose as specification language [Agyed01].

This approach has several benefits. First, application engineers may not have to learn new languages and tool interfaces to write DSL programs. For example, PowerPoint users have no trouble using our Design Editor immediately after understanding what the pallet of icons means in the problem domain. Even if an engineer does not know how to use the tool, learning it can be a useful skill for furthering one's career. Not only is less investment in infrastructure needed, but also the functionality provided by the COTS tool is almost always better than one could imagine providing oneself. After all, the tool provider is an expert at the domain supported by the tool: graphics, databases, forms design, spreadsheet specification, etc. Incorporating such tools leverages all of that expertise.

On the other hand there are drawbacks and things to watch out for with hybrid techniques. The most obvious is the problem of what to do when the tools change that you have relied on to build your approach. There have been suggestions [Copenhafer99] on how to deal with this, but we ourselves have not found it to be a problem so far, going from PowerPoint, to PowerPoint 97, to PowerPoint 2000, to PowerPoint XP. The greater risk is that a less-than-coherent interface to the functionality is presented to users not able to cope with such technological solutions; a stovepipe, monolithic set of support functionality tuned to the user's domain is ideal at first. So the hybrid approach can appear rather intimidating and is somewhat difficult to motivate until the users become familiar with it and see how increased flexibility really can be an advantage for evolving DSL support.

6. Conclusions and Future Directions

I hope that the table in Section 5 can be used as a guide to how to approach a DSL support problem. Only part of this support will involve a "language;" the four aspects of language support - structure, relationships, constraints, and behavior — may require quite uneven treatment in any specific situation. The nature of the analyzers to be provided will almost certainly be important. For example, if graphical support is a major concern, and expressing constraints is not too important, one might choose to use PowerPoint with a COM linkage to Haskell, where analyzers can check preprogrammed constraints. The table should be taken with a (large) grain of salt! This is my broad view based on very shallow knowledge of several of these tools.

One should probably start with pragmatic concerns to see if different approaches are already tied down — must you link with a database anyway, suggesting Access or a particular database package already in use — or is multiple platform evaluation necessary, suggesting Java or Haskell for at least part of the task. Then evaluate the potential tools, looking for holes in the coverage of the rows of the table. Determine if these rows may be of any importance. If so, find interchange mechanisms to access compatible tools that do a good job in that row. I am not well-versed in all the varieties of interchange mechanisms, XML-processing tools, syntax-based tools, generic programming approaches, GUI generators, etc., to provide a thorough analysis here, and even if I were, it would be obsolete next year. So this should simply be a suggestive starting point for your particular DSL needs.

I hope to have convinced the reader of three things:

- That domain-specific languages do not necessarily have anything to do with programming languages at all.
- That designing domain-specific support does not necessarily entail language design in the conventional sense.
- And, that a variety of approaches can be used to tailor domain-specific support to an organization's pragmatic constraints.

Acknowledgements

I would like to thank Jan Heering, Ralf Lämmel, Marjan Mernik, and another very helpful, but anonymous referee for carefully reading the drafts of this paper. I would also like to thank the many members of our research group led by Bob Balzer for over 25 years, formerly at USC / Information Sciences Institute and now at Teknowledge Corp., for the foundations for these observations. And I would like to thank my colleagues Neil Goldman, implementer of the Design Editor, and Martin Feather for myriad fruitful discussions of DSL approaches and support issues.

References

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley: Reading, MA, 1986.

[2] Agyed and D. Wile, Statechart Simulator for Modeling Architectural Dynamics, in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, Amsterdam, Aug 2001, 87–96.

[3] I. Attalli, C. Courbis, P. Degenne, A. Fau, D. Parigot, C. Pasquier, SmartTools: A Generator of Interactive Environment Tools, *10th International Conference on Compiler Construction, CC'2001*, Lecture Notes in Computer Science, vol. 2027, pp. 355–360, 2001.

[4] R. Balzer, M. Feather, N. Goldman, and D. Wile, *Domain-specific Notations for Command and Control Message Passing*, Internal report: USC/Information Sciences Institute, Marina del Rey, CA 1994.

[5] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

[6] J. Bowen, Formal Methods, `http://www.afm.sbu.ac.uk/`

[7] M. van den Brand, A. van Deursen, P. Klint, A. S. Klusener and E. van der Meulen, *Industrial applications of ASF+SDF*, CWI Computer Science/Department of Software Technology Report CS-R9622, 1996.

[8] M. van den Brand, A. Van Deursen, J. Heering, H. A. De Jong, M. de Jong, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, J. Visser, The ASF+SDF Meta-environment: A Component-Based Language Development Environment, *10th International Conference on Compiler Construction, CC'2001*, Lecture Notes in Computer Science, vol. 2027, pp. 365–370, 2001.

[9] L. Cardelli and R. Davies, Sevice combinators for web computing, in *IEEE TSE Special Issue on Domain-Specific Languages,*, C. Ramming and D. Wile, eds. May/June 1999, pp. 309–316.

[10] W. Carlson, P. Hudak, and M. Jones, *An experiment using Haskell to prototype "Geometric Region Servers" for Navy Command and Control*, Intermetrics Research Report, Nov 1993.

[11] S. Chandra, B. Richards, and J. Larus, Teapot: a domain-specific language for writing cache coherence protocols, in *IEEE TSE Special Issue on Domain-Specific Languages*, C. Ramming and D. Wile, eds. May/June 1999, pp. 317–333.

[12] M. Copenhafer and K. Sullivan, Exploration Harnesses: Tool-Supported Interactive Discovery of Commercial Component Properties, in *Proceedings of the Automated Software Engineering Conference*, Cocoa Beach, FL, Oct 1999.

[13] E. Dashofy, A. van der Hoek, and R. Taylor, A Highly-Extensible, XML-Based Architecture Description Language, in *Proceedings of The Working IEEE/IFIP Conference on Software Architecture*, Amsterdam, The Netherlands, August 2001.

[14] Elliot, An embedded modeling language approach to interactive 3D and multimedia animation, in *IEEE TSE Special Issue on Domain-Specific Languages*, C. Ramming and D. Wile, eds. May/June 1999, p. 291–308.

[15] Feather, M.S., A survey and classification of some program transformation approaches and techniques, Meertens, L.G.L.T. ed., *Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation*, Bad Toelz, North-Holland, 1986, 165–195.

[16] D. Garlan, R. Monroe, and D. Wile, Architectural descriptions of component-based systems, in *Foundations of Component-Based Systems*, Gary Leavens and Murali Sitaraman, eds., Kluwer, 2000. (See also: `http//www.cs.cmu.edu/~acme/`)

[17] N. Goldman and R. Balzer, The ISI Visual Design Editor Generator, *IEEE Symposium on Visual Languages*, Tokyo, Sep 1999, 20–27.

[18] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley Publishing Co., Inc., Reading, Mass., 1996.

[19] J. Heering, P. Hendriks, P. Klint and J. Rekers, The Syntax Definition Formalism SDF, *ACM SIGPLAN Notices* 24(11) 43–75, 1989.

[20] J. Heering and P. Klint, Semantics of programming languages: A tool-oriented approach, *ACM SIGPLAN Notices* 35(3) March 2000, 39–48; `http://www.cwi.nl/~jan/semantics/semantics.html.`

[21] P. Hudak, Simon Peyton Jones, and Phillip Wadler, *Report on the programming language Haskell*, Yale Computer Science report YALEU/DCS/RR–777, 1992.

[22] P. Hudak, *The Haskell School of Expression – Learning Functional Programming through Multimedia*, Cambridge University Press, New York, 2000.

[23] P. Hudak and J. Berger, A Model of Performance, Interaction, and Improvisation, *In Proceedings of International Computer Music Conference*, International Computer Music Association, 1995.

[24] J. Jeuring and D. Swierstra, Constructing functional programs for grammar analysis problems, in *Proceedings of a Conference on Functional Programming Languages and Computer Architecture*, La Jolla, CA, 1995, 259–269.

[25] R. Kieburtz, F. Bellegarde, J. Bell, J. Hook, J. Lewis, D. Oliva, T. Sheard, T. Walton, and T. Zhou, *Calculating Software Generators from Solution Specifications*, Technical Report # CS/E-94-032B of the Oregon Graduate Center 1994.

[26] N. Klarlund and M. Schwartzbach, A domain-specific language for regular sets of strings and trees, in *IEEE TSE Special Issue on Domain-Specific Languages*, C. Ramming and D. Wile, eds. May/June 1999, p. 378–386.

[27] P. Klint, A meta-environment for generating programming environments, *ACM Transactions on Software Engineering and Methodology,* 2(2):176–201, 1993.

[28] R. Lämmel and G. Riedewald, Prological Language Processing, eds. M. van den Brand and D. Parigot, in *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, Genova, Italy, Elsevier Science, ENTCS series, 44(2), Apr 2001.

[29] P. E. M. Lopez, Generic parser combinators, in the *2nd Latin-American Conference on Functional Programming (CLaPF)*, La Plata, Argentina, Oct 1997.

[30] Microsoft Access, `http://www.microsoft.com/office/access`

[31] Microsoft PowerPoint, `http://www.microsoft.com/office/powerpoint/default.htm.`

[32] M. Mernik, M. Lenic, E. Avdicausevic, V. Zumer, Compiler/Interpreter Generator System LISA, CD ROM, *Proceedings of the 33rd Hawaii International Conference on System Sciences*, 2000.

[33] M. MERNIK, M. LENIČ, E. AVDIČAUŠEVIĆ, V. ŽUMER, Multiple attribute grammar inheritance, *Informatica*, vol. 24, no. 3, pp. 319–328, 2000.

[34] C. Ramming and D. Wile, eds., *IEEE TSE Special Issue on Domain-Specific Languages*, May/June 1999

[35] T. REPS AND T. TEITELBAUM, *The Synthesizer Generator*, Springer-Verlag, New York 1988.

[36] D. SHARNOFF, *Free Compilers:* `http://www.idiom.com/free-compilers/`, 2001.

[37] I. BEN-SHAUL AND G. KAISER, *A Paradigm for Decentralized Process Modeling*, Kluwer Academic Publishers, Boston, 1995.

[38] Sun Microsystems Forte, `http://www.sun.com/forte/ffj/`

[39] M. TALLIS, N. GOLDMAN, AND R. BALZER, The Briefing Associate: a role for COTS applications in the Semantic Web, *International Semantic Web Working Symposium (SWWS)*, Stanford, California, U.S.A. Jul, 2001.

[40] S. THIBAULT, R. MARLET AND C. CONSEL, Domain-specific languages: from design to implementation application to video device drivers generation, in *IEEE TSE Special Issue on Domain-Specific Languages*, C. Ramming and D. Wile, eds. May/June 1999. p. 363–377.

[41] D. WILE, *Popart: Producers of Parsers and Related Tools*, Reference Manual, USC/Information Sciences Institute, Marina del Rey, CA 1993.

[42] D. WILE, Local Formalisms: Widening the Spectrum of Wide-Spectrum Languages. Meertens, L.G.L.T. ed., *Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation*, Bad Toelz, North-Holland, 1986, 459-481.

[43] D. WILE AND R. BALZER, *Survey Instrument Creator (SIC) Training Manual*, Teknowledge Corp. 2000.

[44] XMLSoftware. `http://www.xmlsoftware.com/`, 2001.

Received: September, 2001
Revised: November, 2001
Accepted: November, 2001

Contact address:
David Wile
Teknowledge Corp., Suite 231
4640 Admiralty Way
Marina del Rey, CA 90292, USA
Phone: +1-310-578-5350
Fax: +1-310-578-5710
e-mail: `dwile@teknowledge.com`

DAVID WILE received his Sc.B. degree in Applied Mathematics from Brown University in 1967 and his Ph.D. in Computer Science from Carnegie-Mellon University in 1974. In 1973 he joined the University of Southern California's Information Sciences Institute, where he became a project leader. His early interests were the formal specification of functional and non-functional requirements and user-guided transformation of specifications into implementations. Dr. Wile is interested in many aspects of the programming process, from language designs for problem specification and programming, through the specification and optimization design processes, to meta-programming environments used to realize these designs. His recent research interests include: adaptation of COTS tools for the specification and implementation of domain-specific languages, developing formal specification languages for software architectures, and the use of architectural specifications in requirements engineering.

Dr. Wile is a Senior Research Scientist in Teknowledge Corporation. He was previously a Research Professor at the Computer Science Department of the University of Southern California. He was Program Chair of the Foundations of Software Engineering Conference, is an ex-editor for IEEE Transactions on Software Engineering, and will be Program Co-chair for the Automated Software Engineering Conference in 2002. He is a member of the IEEE, the ACM, SIGPLAN, SIGSOFT, Sigma Xi, and IFIP's Working Group 2.1 on "Algorithmic Languages and Calculi."