Next Generation Modeling and Observation Tools for Hard Science Linguistics

Jack Myers, Rowan University, Department of Computer Science, U.S.A.
Robert Hussey, Rowan University, Department of Computer Science, U.S.A.
Kevin Desmond, Rowan University, Department of Computer Science, U.S.A.
Saroash Liaqat, Rowan University, Department of Engineering, U.S.A.
Zabih Shinwari, Rowan University, Department of Computer Science, U.S.A.
Louis Szgalsky, Rowan University, Department of Computer Science, U.S.A.
Douglas Taggart, Rowan University, Department of Computer Science, U.S.A.

ABSTRACT

Hard science linguistics (HSL) is unique among linguistic programs in that it studies communicative behavior of individuals in the real world, rather than theoretical, non-scientifically observable domains. Its founder, former physicist and early computer scientist Victor Yngve, applied his experience with the sciences, utilizing the principles of systems, boundaries, and control procedures to define this branch of linguistics. HSL has also developed its own expressive modeling language, replete with properties, tasks and linkages. Given the scientific discipline of this systems-based approach, there is remarkable overlap with the domain of computer science. However, the paucity of computer applications for the hard science linguist is both unexpected and unfortunate. Object-oriented programming provides a mechanism to translate linguistic elements into computerized models. In this article, we demonstrate how to leverage advances in object-oriented and prototype-based design to construct dynamic and reusable HSL toolkits.

Keywords: *Hard Science Linguistics; Modeling and Simulation; Object-Oriented Design; Observation Tools; Groovy*

INTRODUCTION

Hard Science Linguistics (HSL) takes a scientific approach to linguistics and contends that by analyzing the real life observable context of communications, one can understand and model the nature of such communications. HSL imbues all linguistic analysis and modeling with a systematic approach that lends itself well to computerization. As a relatively new discipline, it has not profited from computerization (Sypniewski, in press) to the same degree as the natural language discipline which has been explored by cognitive scientists in the development of computerized learning systems

(Graesser, Hu & McNamara, 2005; Duan & Cruz, 2011).

The front end of HSL has been well established and tested, yet the back end (HSL computerization) remains relatively unexplored aside from some early modeling using Python, Prolog and the procedural language Forth (Sypniewski, in press). The design principles described in this article lay the groundwork for the implementation of useful tools for today's hard science linguists. This research offers the computer scientist a subject area ripe for the development of sophisticated modeling tools while placing powerful theoretical and practical utilities in the hands of linguists.

The need for modeling and simulation tools

The field of Hard Science Linguistics has seen a surge of popularity in the past few years, as more and more academics and researchers are focusing on the field of human interaction and communication. Much of HSL makes use of constructing a model indicative of an interaction (a linkage) and comparing it to similar interactions. Computerizing these processes would enable researchers to construct linkages in a simpler, more uniform manner and test them thoroughly.

By bringing creation and testing of linkages into a more uniform and computerized form, HSL researchers can more easily share ideas and findings with each other. Additionally a computerized simulation tool can become a simple and easy-to-use utility, especially for those researchers not overly familiar with software applications.

The need for field observation tools

Today's information laden society could be considered obsessed with recording observations. The number of web blogs has been steadily rising, reality television has primed audiences to be comfortable observing the private lives of others, and YouTube features many channels devoted to recording daily video diaries.

Such an environment might be considered a boon for hard science linguists who document communicative behavior. However while society has become more receptive to observers, ironically, it has also become increasingly sensitive to such observations. In particular, observing communicative behavior in public places often elicits suspicious reactions as businesses wonder why they are being observed and for what purpose. Concerns abound on where the information will land and how the information may be used. Our challenge is enabling the hard science linguist to observe corporate communications unobtrusively such that the effects of observations on communications are reduced to near zero levels.

HSL STRUCTURE

Before exploring our computerized model of this area, it is worth summarizing key HSL concepts.

Systems and Linkages

A *system* has primarily the same meaning to a hard science linguist as it would to a computer scientist. In computer science, a system defines the boundaries of the application area to be computerized. In HSL, a system describes a manageably sized scope of observations. They are denoted by including the system name in square brackets.

A *linkage* is one form of an HSL system. It reduces the elements of a group of people and their linguistically relevant surroundings to those pertinent to the explanation of communicative behavior. A linkage, therefore, is a container of additional systems. Not every action a person makes while communicating is important to a communication. Not every aspect of the surrounding is relevant either. The linkage sets the proper boundaries to model the proper components.

One aspect of a linkage that may significantly alter the meaning of a conversation is the setting (Sypniewski, 2008). To illustrate, two women may be having a conversation and one says to the other, "Here comes the wolf now." Even though the words are identical, the interpretation would be vastly dissimilar in two different settings. At a cocktail party, "wolf" may be interpreted as a slang word for a womanizer; in a zoological setting, the comment likely refers to an actual wolf. Settings are not always important to include in a linkage, and there are instances when a linkage may take place in multiple settings as action shifts from one place to another.

Every system is further described by a number of linguistic *properties*; systems capable of performing actions are assigned various *tasks* that they can perform. Tasks as well as linguistic property changes are denoted in angle brackets. Yngve (1996) defines linguistic properties as "theoretical constructs characterizing people from the point of view of how they communicate." Tasks have been characterized as "descriptions of linguistically relevant behavior, somewhat analogous to functions in computer programming" (Sypniewski, 2001).

As the term "property" has two contexts in this article, subsequent references to an object-oriented property will be known as "property" and their HSL counterpart will be known as "linguistic property." An obvious path forward would be the correlation of linguistic property to the object-oriented property, and the HSL task to the object-oriented method.

Participants and Role Parts

Important parts of a linkage are the participants who engage in discourse. A participant is a *representation* of a person in much the same way as an object-oriented class is a representation of a real world object.

An important part of modeling a participant in a computer simulation is the ability to tie a participant to its corresponding role parts. A role part is the functional aspect of a participant (Sypniewski, 2010a); it encapsulates a set of repeatable behaviors that correspond to an aspect of a participant's communicative behavior.

The meaning of a conversation can vary according to the particular role that an individual is playing at the time. For an example, consider an adult conversation between two friends – Ken and Adam – on a sexual topic. If Ken's six year old daughter were to enter the context of the conversation, Ken may choose to substitute less explicit words or increase the use of eye contact or gestures to convey his meaning. The participant Ken began the conversation with the role part of *friend*, but mid-stream continued the conversation playing the role parts of both *friend* and *father.* This is linguistically significant, and it is necessary to keep track of role parts as they change throughout a linkage.

Role parts are often understood in the context of plays or movies. The role of the Joker in the Batman movies and the 1960s television series was adeptly played by Jack Nicholson, Heath Ledger and Cesar Romero. There were certain communicative behaviors one would expect of the Joker, for instance [Joker]<taunts Batman>. In fact, all three versions of the Joker engage in similar interactions, but Romero's Joker was the most comic rendition, whereas Ledger's was the most sinister. Nicholson's Joker, albeit brilliant, was sometimes criticized for being equal parts Joker and "Jack Nicholson" (who some might argue has become so caricatured as to be both a role part and a participant). Yet from a modeling point of view, unless we focus our assemblages on the fine arts, we are more likely to see instances like participant [Ken] who plays dual roles as father and friend.

HSL requires capturing both participant and role part data. We initially ruled out having a role part be a simple string property of a participant. Even though Ken could be assigned the role part properties of "friend" and "father" in an associative array, this would be unsatisfactory based on object-oriented principles. One such principle, developed by object-oriented pioneer Bertrand Meyer, is Modular Composability, which calls for "the production of software elements which may then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed" (Meyer, 42). Relegating the role part to a string would violate this criterion, as a property could no longer be shared across various types of objects. However, a role part can be a property of a participant as its own object type. In this way, unlike simple value-based properties, it can contain its own properties and methods.

This makes linguistic sense. Ken can be allowed to have methods and properties relevant to the linkage at the *participant* level. For example, Ken might have the linguistic property of anxiousness with values "in a hurry" or "not rushed"; his tasks (i.e., methods) might include smile() or laugh(). But the *role* of father might introduce more specific properties and methods for participant [Ken]. [Ken the Father] might have an expectation of a topic's sensitivity to children – a linguistic property that would not be manifest in his discussion with his friend were his daughter not present. If a participant were able to assume and discard multiple role parts, it would in effect be expanding or contracting its set of relevant linguistic properties and tasks.

We also considered modeling role parts as a subclass of a participant, which is appealing from an object-oriented modeling viewpoint. [Ken the Father] could be a role part subclass of [Ken] the participant. As it would inherit all of the properties and methods of [Ken], [Ken the Father] would have access to its own elements as well as the elements of its parent object. However, Ken could be assuming multiple roles as in our example. If we allow [Ken the Friend] to be an additional subclass of [Ken] we force our object to be *either* [Ken the Father] or [Ken the Friend].

This would only be valid if a participant only played one role at a time. The model would hold if Ken's dialog with his child was observed as [Ken the Father], and his dialog with his friend Adam was captured as [Ken the Friend]. However, it is possible that Ken's dialog with his friend is influenced by the second role Ken is playing – even though he might not be communicating with his daughter at that instant.

The participant / role part inheritance structure could be flipped. Participant Ken could multiply inherit from role parts "Father" and "Friend." While conceivable in programming languages that support true multiple inheritance such as Eiffel, this design complicates the model greatly, limits the choice of implementation languages, and introduces the complexity of contextual de-inheritance.

Therefore, we abandoned inheritance for role parts, instead creating the participant class with a property "RoleParts" – an array of separate role part objects.

Props and Channels

A prop is a model of an object in the assemblage that is relevant to the communications (Yngve, 1996). Only relevant linguistic properties need to be recorded. Consider a soccer referee's interaction with a player. The color of the prop he holds (a yellow card or a red card) is linguistically significant to the conversation, whereas the material of the card may not be.

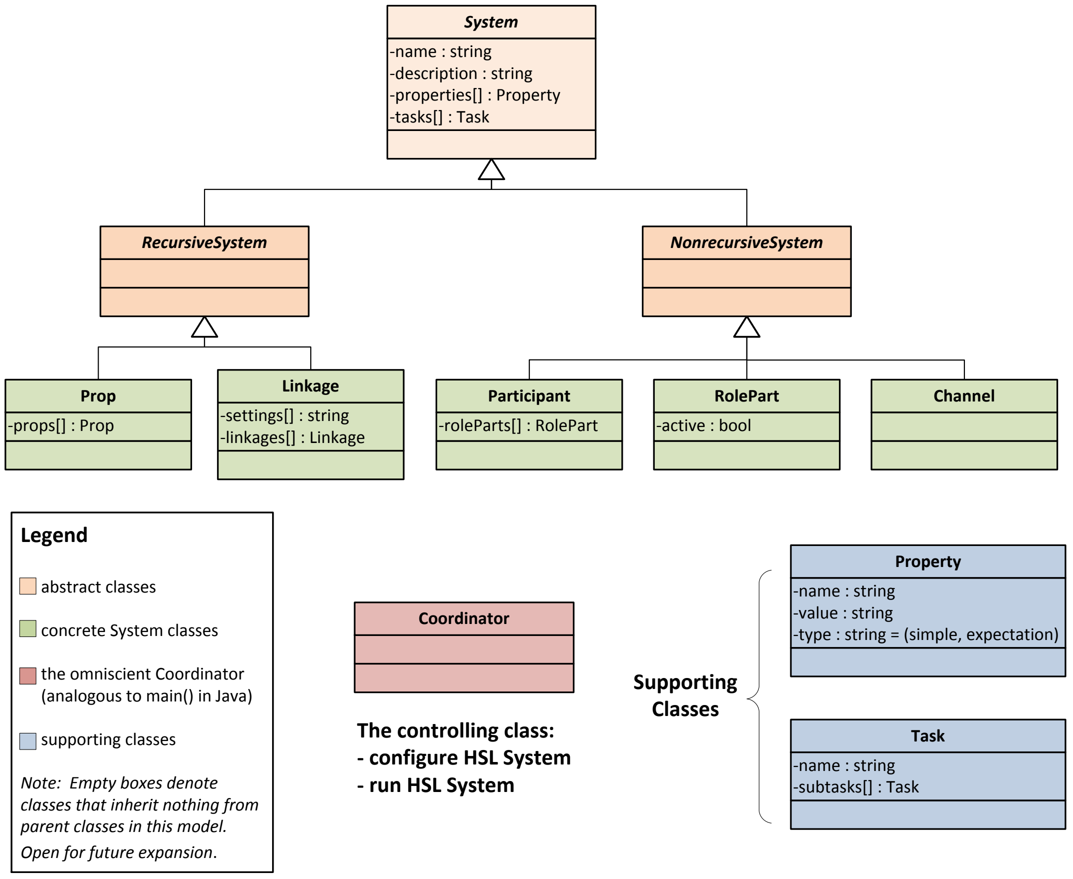
A prop's owner might not be linguistically relevant in all linkages, however it might be in the case above. Imagine if the soccer player held up his own red card to the referee! The red card has a distinct linguistic significance when associated with a particular participant playing the role part of referee.

Props are fairly straightforward to implement, as they requires very little functionality. Implementing a prop, and to a larger extent a linkage, should be done using object-oriented design principles as objects in an object-oriented design are mostly defined by their relationship between other objects, and not so much by their functionality.

In the implementation of a prop, it is important to be able to preserve the relationship between the prop and the system with which it interacts. For example, in the above scenario the role part [referee] would need to be able to access and interact with the prop [red card]. This can be implemented fairly simply in multiple ways. An object representing a system, like [referee], could have access to a list of props via stored references.

The last system to be modeled is a *channel*, which defines an energy flow whose representation is limited to the parts essential to the linkage. Examples of energy flow include the firmness with which a participant shakes hands or the intensity of a knock on the door. Thus channels may have associated linguistic properties and values, such as <grip/firm> and <intensity/loud>. Channels may also have associated tasks, such as <increase volume>.

Figure 1. Object Model of HSL Constructs



BUILDING A MODELING TOOL

The first tool we propose is a modeling / simulation application to establish and exercise linkages.

A Sample Linkage

Yngve (2004) formalized the recording of observations into a standard structure as seen in Figure 2. The example captures an interaction between a deli worker and a customer. There are two role parts: worker and customer, as played by two participating individuals identified as "worker 1" and "customer A." In the example, there is also an LED monitor which displays a number. The value of the LED monitor is a property, which is initially set to 43.

Figure 2. Communicative Behavior in a Deli Displayed Using Linkage Notation

```
[LED monitor]<value/43>
[worker 1]<calls number 44>
[worker 1]<calls number 45>
[LED monitor]<value/45>
[worker 1]<asks what customer wants>
[customer  A]<orders sliceable item(s)>
[worker 1]<slices item>
[worker 1]<weighs item>
[worker 1]<asks if weight is okay>
[customer  A]<indicates yes>
[worker 1]<asks if anything else needed>
[customer  A]<indicates no>
[worker 1]<delivers closing statement>
```

There is also a less commonly used notation which resembles circuit diagrams, but this is less amenable for computerization.

Developing a reusable model

Linkages, like the example above, as well as participants, role parts, channels and props are each a type of system and need to be instantiated for each HSL model.

Certain atomic systems can be classified as non-recursive systems as they will not act as container classes for themselves. A participant could not contain additional sub-participants, but linkages may be composed of subordinate linkages and props may contain subordinate props, such as a driver's license contained in a participant's wallet. Hence, both Linkage and Prop will be considered to be recursive systems.

At this stage, our model does not manifest different properties or methods for recursive and non-recursive systems, but the distinction was incorporated to allow for future development. Classes for System, NonrecursiveSystem and RecursiveSystem will be implemented as abstract classes, as their instantiation would not be required by the linguist/modeler.

Both methods and properties in our model will be either *intrinsic* or *dynamic.* Intrinsic methods and properties would be built directly into the appropriate concrete System classes.

Intrinsic properties will include name and description – declared at the System level. Each concrete class will include an identifying name and a longer description property for documentation purposes. Other intrinsic properties include the setting(s) of a linkage and the status of whether a particular role part is active or inactive. However, in order to make the model reusable across linguistic scenarios, most properties will be dynamic. An associative array of properties will be inherited by all objects in the system. In this way, each instance of a class will be able to store uniquely named properties and their corresponding values.

Methods in this model will likewise be *intrinsic* or *dynamic.* Dynamic methods are those that are relevant to a particular system, and will be members of the array tasks[] which is inherited from System. Intrinsic methods will also be used, such as getPropertyValue which will return the value of any property name that is passed as a parameter. Intrinisic properties will require intrinsic getter and setter methods. Participant will also require unique intrinsic methods for the activation, deactivation, and possibly management of role parts.

As discussed later in the article, task logic is best handled via dynamic methods whose method body is created by a coordinating class. However Task objects are still required to be created for the purposes of associating them with a particular Linkage, and for building a task/subtask hierarchy.

The Coordinator

System objects would have access to the methods and properties of each contained object, e.g., a Participant would have full access to the properties of its RoleParts. However, there is a need for some coordinating object that would control the role parts by adding or removing role parts from the role part array as appropriate to the linkage.

The Coordinator, therefore, was added as an extra theoretical construct not in the current HSL lexicon. This coordinating class is proposed to facilitate the computerization of HSL, especially the coordination of multiple linkages. The Coordinator is responsible for setting up linkages, initializing properties, establishing methods, executing an HSL model, and outputting results in HSL notation.

To perform these functions, the Coordinator requires direct access to the appropriate members of the System classes. For example, the Coordinator must be able to either access a Participant's collection of role parts directly. Per the previous example of Ken and Adam, the Coordinator will be able to assign a new role part to participant [Ken] by simply pushing the new role part onto Ken's collection of role parts. The code to add and remove role parts can be simple.

```
ken.roleparts.add('Friend of Adam')
ken.roleparts.remove('Friend of Adam')
```

The preceding approach may be easy to implement, but it can cause issues if later it is decided that some code must run each time a new role part is added to a participant role part collection. To remedy this and make future logic changes with regards to system data member manipulation, the code for adding and removing role parts as well as any other data member associated with a system can be encapsulated in a method. With this approach, the previous examples are reconfigured.

```
ken.addRolePart('Friend of Adam')
ken.removeRolePart('Friend of Adam')
```

Both addRolePart and removeRolePart would be intrinsic methods implemented in the participant class, and similar methods would be implemented in the appropriate classes for other data members.

The Coordinator is also responsible for executing the linkage once it has been properly set up by receiving explicit instructions from the user. These instructions could be similar to the format of a linkage, in that a top-level task could be given, followed by other tasks, assignments of linguistic property values, role part changes and other directives. The order in which these individual linkage steps are given would be the order in which the Coordinator executes them.

To give another example, first suppose that we are in a state in which the user has specified the systems associated with the linkage, as well as the initial values (such as role parts, linguistic properties and tasks) already. This particular linkage describes participant [Jo] who opens her door, sees a parade on her street that she was not expecting, and she exclaims in surprise. Opening the door is the top-level task and that this task causes the door's linguistic property open to be "yes." The next step specified is the task in which Jo sees a parade, and this task causes Jo's linguistic property surprised to become "yes."

In this simple example, the Coordinator would begin by executing the top-level "openDoor" task on the jo object. When the user defined this task using the Coordinator, a linguistic property value change was specified as the result of the task. As such, the "open" property of the door prop object would be set to "yes" in the execution of the task. The Coordinator, having executed the top-level task, will now move on to the next step specified, which in this case is another task called "seeParade." Once again, the user has defined a linguistic property value change as the result of the task, and as such the "surprised" property for Jo would be set to "yes" automatically via the task execution.

Modeling role parts and props

Earlier we proposed that a participant object contain an array of role part objects that would indicate which role parts were used by the Participant in the linkage. Each role part, in addition to its inherited fields from the System object, would have two additional fields. The first would be an "active" field, which specifies if the role part is currently in use. The second would be a field to indicate which participants would be affected by the role part. This would most likely take the form of an array, since it is possible for a role part to be relevant to more than one other participant. If we consider a linkage with Jarvis, and his two children Rufus and Jemma, the father role part would be stored in the Jarvis instance. The actual instance of the father role part would contain references to both Rufus and Jemma.

As far as props are concerned, we need to decide whether it is necessary to capture associations between props and participants when they exist. There may be cases in which a participant in a linkage checks the state of a prop. Does the participant need to have an explicit association to this prop prior to the check? The fundamental question around the modeling technique becomes whether the Participant should set its next state (properties, expectations, etc.) or should the Coordinator set these aspects for the Participant? While creating an object-oriented model of prop ownership by a participant who is able to check the values of the prop's properties may be more true to life, in terms of HSL, we believe a coordinator class makes more sense and simplifies matters. While building the model, the Coordinator will know that the Participant needs to check the state of a property. Hence the Coordinator will be able to modify the appropriate data members through methods called on the participant object.

Implementing the model

A key decision is the selection of a candidate language for implementation of the model. Java, while both object-oriented and prevalent, is not an ideal choice. More recent successors to Java, such as the language Groovy, a dynamic language that integrates with Java libraries, handle dynamic methods with ease.

The classic object-oriented approach to methods, and indeed all elements of classes, is to bind them to an instance of an object upon creation. In other words, the sum total of properties and methods is statically fixed at the time an instance of a class is instantiated. This method, employed by Java, is not well suited to a model which requires methods and properties to be created at runtime.

This limitation of Java was described in Brent Carlson's patent (US Patent No. 6,766,324 B2, 2004), which outlines how attributes and methods can be added or removed during runtime using configurable Java classes. Yet such an approach is more unwieldy than desired for a modeling application, requiring the use of special libraries to compensate for the lack of innate Java functionality. Subsequent efforts to make traditional object-oriented languages more dynamic are better suited to our task.

Groovy classes can extend Java classes and support runtime meta-programming which allows the synthesis of methods using a meta-object protocol (Layka, Judd, Nusairat and Shingler, 2012). Groovy allows a method to be created using an ExpandoMetaClass which is invisible to the user (Richardson, 2009). Groovy has a simple Java-like syntax as seen in the sample code which creates and invokes the "see parade" task described earlier.

Figure 3. Sample Groovy Code to Dynamically Create the "See Parade" Method

```
// Participant jo previously created

// Create dynamic method
jo.metaClass.seeParade = {
   println "[" + this.name + "]" +
    "<sees parade>"
   this.properties['surprised'] = "yes"
   println "[" + this.name + "]" +
    "<surprised/" + this.surprised + ">"
}

// Invoke new method
jo.seeParade()
```

In keeping with our design pattern, a Task object "seeParade" would also be created and its properties set. However such actions would be more for HSL documentation purposes rather than method execution.

Figure 4. Code from an Instance of Coordinator that Creates an HSL Linkage

```
// Instantiate objects and properties
parade = new Linkage('Parade', 'Jo sees a
parade')
parade.addSetting('street hosting a parade')
jo = new Participant('Jo', 'average Jo')
door = new Prop('door', 'Jo\'s
front door')
audience = new RolePart('audience', 'viewer of
parade')
surprised = new Property('surprised', 'no',
'simple')
jo.addProperty(surprised)
open = new Property('open', 'no', 'simple')
door.addProperty(open)

// Instantiate tasks and methods
walkToDoor = new Task('walkToDoor')
jo.metaClass.walkToDoor = {
   println "[" + this.name + "]" +
    "<walks to door>"
}
openDoor = new Task('openDoor')
openDoor.addSubTask(walkToDoor)
jo.metaClass.openDoor = { [Door] door ->
   this.walkToDoor()
   println "[" + this.name + "]" +
    "<opens door>"
   door.properties['open'] = "yes"
   println "[" + door.getName() + "]" +
    "<open/" + door.properties['open'] +
    ">"
}
applaud = new Task('applaud')
audience.metaClass.applaud = {
    [Participant] participant ->
    println "[" + participant + " as "
     this.name + "]<applauds>"
}
seeParade = new Task('seeParade')
jo.metaClass.seeParade = {
   [RolePart] audience ->
   println "[" + this.name + "]" +
    "<sees parade>"
   this.properties['surprised'] = "yes"
   println "[" + this.name + "]" +
    "<surprised/" + this.surprised + ">"
   jo.addRolePart(audience, 'active')
   jo.rolepart['audience'].applaud(this)
}
```

Note that the seeParade method has been enhanced to include the addition of the role part "audience" and its corresponding task "applaud." The code also illustrates how intrinsic (e.g., addSubtask) and dynamic (e.g., applaud) methods are utilized during model creation.

There are only two actual steps in this linkage (opening the door and seeing the parade). Both steps actually have a number of effects, which could be considered sub-steps.

Table 1. Executing a Linkage

Commands Invoked by Coordinator	Effects
jo.openDoor(door)	[Jo]<walks to door> [Jo]<opens door> [door]<open/yes>
jo.seeParade()	[Jo]<sees parade> [Jo]<surprised/yes> [Jo as audience]<applauds>

As shown in Table 1 above, first the openDoor task is called on the participant [Jo]. This task has a subtask which has no effect other than outputting itself in HSL notation. After calling this subtask, the openDoor task’s logic is executed, which sets the open property of the door prop to 'yes.'

After opening the door, Jo sees the parade outside, represented by the seeParade task called on the participant [Jo]. Via execution of this task, Jo gets a new and active role part called 'audience,' which could have been a shared role part if there were other participants in this linkage. The other effect of the seeParade task is to set the surprised property of the participant Jo to 'yes.'

Model Flexibility

After conceiving our model using typical HSL examples, we wanted to explore its robustness using complex linkages and two concepts very challenging to model: expectations and orthoconcepts.

To evaluate the strength of our object-oriented model, we tested it against a very complex linkage based on a typical office day that includes four separate sub-linkages. One situation that needed addressing turned out to not put any stress on the model. Circumstances in the complex linkage called for a single task from one of the linkages to occur much sooner than the rest of that linkage. This formulated the question of whether or not *partial* linkages can be defined. As it turns out, since the Coordinator sets up every object in the system, this did not really matter. If the entire linkage is defined beforehand, the single needed task can still be called. On the other hand, it is perfectly feasible for the Coordinator to only partially define a linkage at one point in time, and then add more to that linkage later.

However, there were two facets of the complex linkage that did not hold up as well. Participants Jack and Lisa were talking to each other about a budget. This is defined by two separate but parallel tasks, [Jack]<talk to Lisa about budget> and [Lisa]<talk to Jack about budget>. In our model, there is no means of running these tasks simultaneously. Instead they would occur as if the tasks were accomplished sequentially. Hence, the computer output in this case is not a true representative of a real linguistic behavior and this needs to be further developed

A second area of difficulty was caused by one linkage affecting another. In the linkage there existed an "interruption behavior" during which the conversation between the two participants is paused, and one of the participants gets involved into a second defined task. The easiest way around this situation is to rely on the Coordinator to handle it; however, it might be an improvement to encapsulate interruption behavior within a linkage itself.

We next considered how to model expectations. In a communicative exchange, some properties are harder to observe. For example, in the "Jo sees a parade" linkage, Jo has a relevant expectation that there will be nothing out of the ordinary happening on her street when she opens her door. While not directly observable, certain outward expressions of Jo's surprise could support the fact that her expectations were not met. This could be modeled using an expectation and expectation procedure rather than just a simple property for the surprise of seeing the parade. Since Jo doesn’t expect to see a parade, the participant object which represents Jo could have been given an expectation as shown below.

```
expectEverydayActivity = new Property
   ('expect everyday activity outside','',
'expectation')
```

This defines a new expectation, using a specific case of the Property class (‘expectation’), and specifically says that the system associated with the expectation expects everyday activity to be occurring outside. The effect of seeing the parade (seeParade task) could then set the expectation value to '-', which means the expectation has been contradicted or not met.

Also, based on this expectation, expectation procedures could be defined as follows:

```
[Jo]<expect everyday activity outside>::
   [Jo]<go to work>

[Jo]<-expect everyday activity outside>::
   [Jo]<watch unusual outside activity>
```

The above HSL notation should be read as a pairing of pre- and post-conditions, i.e., as two separate if/then statements. The expectation procedure would define a branch in the logic of the linkage in which the next step would depend on the results of the previous steps (Sypniewski, 2010a). Specifically, if at this point in time, Jo's "expect everyday activity outside" expectation is not contradicted, then Jo will go to work. If however the expectation is contradicted prior to this point in time, then Jo will stop and watch the unusual outside activity. Furthermore, the expectation and expectation procedure could be used in conjunction with the surprised property already in use to further describe the effect of seeing the parade.

Additionally, due to the use of system objects which have a collection of associated properties independent of the linkage currently being executed, there is the benefit of properties (and therefore expectations) being maintained across linkages. For example, when the expectation "expect everyday activity outside" is associated with the participant Jo, this expectation and value is stored as part of the Jo object. This expectation-value pair can then be manipulated during the currently executing linkage, and if there is another linkage either as a child linkage of the first or simply a separate linkage in which the participant Jo plays a part, the same property-value pair will remain set and associated with the Jo object. This can be a benefit in certain scenarios, such as when there are multiple linkages interacting with one another and it is assumed the constituents of each linkage will be shared across linkages with the same set of expectations and other properties for each constituent. Thus, dealing with expectations *in their simplest form* is well covered by our model.

The final test our model was the use of HSL orthoconcepts to handle entities which are not tangible objects in the strictest sense. Orthoconcepts are a relatively new addition to hard science linguistics, created to handle the difficulties in modeling situations where past experiences shape an event. In Lara Burazar's paper on orthoconcepts (2006), she uses the example of a game of tag, in which players take turns "being it." Our computer model can be set up to handle orthoconcepts, provided that we treat orthoconcepts as a specialized form of properties, similarly as was done with expectations. At their essence, orthoconcepts are simply properties who float among the participants in a linkage. In the first iteration of our model, orthoconcepts can be represented by setting the type value of Property.

Outcomes and Future Work

The primary outcomes of our research include the object model described in this article, the approach to the Coordinator class, and the recommendation to code this tool using an object-oriented programming language with dynamic methods. Further exploration and development is required to translate such specifications into a production application.

Moreover, our model could be further refined to handle parallel tasks, interruption behavior, orthoconcepts and expectations. Orthoconcepts, for instance, could be handled by implementing a sub-class of the Property class that would inherit all of the generic Property attributes, while include new logic to handle the jumping from one participant to another.

While our model allows for expectations which are independent of the passing of time, such as the "expect everyday activity outside" expectation, HSL allows for expectations which are time dependent. Our model as-is does not account for such expectations. For example, consider a participant Amber whose phone may ring due to a call from another participant, Della, within a particular time frame, such as 30 minutes from the current time. This can be described in HSL notation as: [Amber]<expect call from Della within 30 minutes>.

Although the previous notation covers only one expectation of Amber, it could potentially change properties of Amber –not only when it is met or contradicted, but also at points of time in between. In other words, Amber might have a property such as "nervous" with a value of "not at all" when the expectation is first produced by Amber and a value of "extremely" after 30 minutes passes and Della has not called. For the time in-between, the value of that property will range from "not at all" to "extremely" based on the amount of time which has passed. The expectation's reliance on time causes it to have non-discrete update intervals for the system which is associated with the expectation.

To fully implement expectations and expectation procedures would require allowing for property and expectation value updates based on non-discrete intervals of time. However, using a polling mechanism, which does work with discrete intervals of time, would allow for a close enough approximation of the functionality of expectations and expectation procedures. This polling mechanism would check to see if an associated expectation has been met, contradicted, or neither after a set amount of time, and based on this check as well as the amount of time that has passed, it may cause property updates appropriate to the situation. To be able to accomplish this, the polling mechanism would need to be a part of the Coordinator, which is able to access the necessary information as well as update the components of the linkage, and the user would need to define the polling interval as well as the effect each interval will have based on the status of the expectation at that time. Although such an implementation would still only allow for discrete updates, it better reflects the effects the passage of time can have due to an expectation.

In terms of programming, this would require either a change to the property class or a separation of property and expectation into two separate classes, both inheriting from the system class. Either way, the class could be modified to store the initial time when an expectation is first activated as well as the interval of time in which the status of the expectation would be checked. Initializing the expectation might look like "expectEverydayActivity = new Expectation('expect everyday activity outside', ' ', time.currentTime(), 0.05)." This would specify the same expectation described previously, except in this case we are specifying that the expectation begun at the current time and will be polled for changes every 50 milliseconds. The Coordinator would also need to be modified to include the polling mechanism, which could be implemented using an event system which fires each update interval.

Another area of development is to build a user-friendly front end to produce the requisite objects for any linkage. The "Jo sees a parade" linkage is focused on one particular linkage and is not generic in the least. It is possible to use the same general approach as described, but be more inclusive. For example, if the user is given a dynamic form enabling them to input individual props, participants, and other systems which are a part of the linkage they are using, then it is possible to take the input from these forms and create the necessary objects just as done previously. Jo would no longer be created as listed in Figure 4, but rather, variables assigned to the fields of the form would need to be used, such as:

```
participant[i] = new Participant
  (formTextBox1.Text, formTextBox2.Text)
```

where i represents the current id of the participant being created and the textboxes would contain "Jo" and "average Jo" respectively.

Two other intriguing continuations to our work are worthy of further exploration. A recent extension to object-oriented development known as "typestate-oriented programming" has been proposed by Carnegie Mellon University. This approach still leverages classes, but objects are allowed to change states within a class and "each state may have its own representation and methods which may transition the object into a new state" (Aldrich, Sunshine, Saini & Sparks).

The use of prototype-based programming languages, such as Self, Io or Slate, also could expand the functionality of our modeling tool. These languages might reduce the programming burden on the linguist. Essentially class-less, new objects are created by cloning existing objects as a starting point. Other research at Carnegie Mellon used the dynamically typed language Slate to model aquatic behavior (Salzman & Aldrich). Like Groovy, methods can be dynamically created and destroyed in Slate. Prototype-based languages could model participants with variant behaviors by creating copies of participant objects and slightly modifying their linguistic properties and tasks. Linguists might also be able to clone a pre-established Coordinator object to easily model what-if scenarios within a particular linkage without an in-depth knowledge of computer programming.

FACILITATING OBSERVATIONS

The second tool we propose is an HSL observation tool.

Design Principles

HSL is based upon observations and experiments in which researchers often observe people's communicative behavior in a natural setting (Sypniewski, 2010b). Observations need to be highly focused on the linguistically relevant aspects of a situation. The observation tool must therefore include not only participants and their actions, but elements of the environment that affect the conversation results (Sypniewski, 2008).

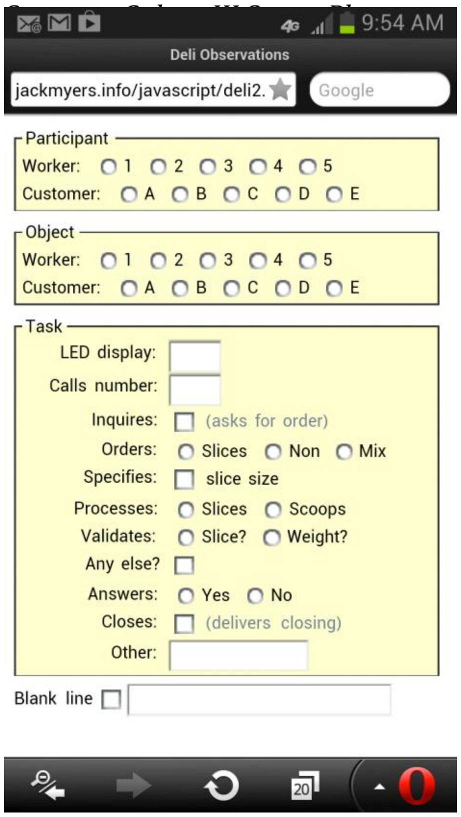
Fortunately, there is an emerging social phenomenon that allows an observer to be unobtrusive – the increasingly normal trend of individuals to stand motionless while they text or email using smart phones. The hard science linguist can take advantage of this trending behavior to conduct less obvious observations of business. The smart phone not only helps the observer blend in with the surroundings, but it often provides an embedded camera useful for capturing key details about the environment.

Minimizing the inputs (e.g. typing, numbers inputs or notations) is critical for recording observations efficiently. Moreover, the interface should effectively use screen real estate and anchor user interactions onto a single screen of a typical smart phone. The HSL observer should be able to limit the observable aspects of a situation to a manageable number of expected participants and behaviors. However, the tool must accommodate unexpected or infrequently occurring observations by allowing the user to capture free form observations.

An implementation model

Figure 5 depicts how the user interface for the deli linkage might appear on a smart phone.
The front end of the observation tool is coded as an HTML form with basic JavaScript. All form elements are divided into three <fieldset> regions for Participant, Object and Task. Text, checkbox and radio button controls in the Task <fieldset> invoke the record method when the value of the control changes as shown in the excerpted code below.

Figure 5. Representation of Observation Tool on a



The front end of the observation tool is coded as an HTML form with basic JavaScript. All form elements are divided into three <fieldset> regions for Participant, Object and Task. Text, checkbox and radio button controls in the Task <fieldset> invoke the record method when the value of the control changes as shown in the excerpted code below.

Figure 6. HTML for Selected Form Controls

```
<fieldset>
 <legend>Task</legend>

  <label for="led">LED display: </label>
  <input id="LED" name="LEDmonitor"
      type="text" size="3"
      class="property"
      onchange="record(this)" /><br />

 <label for="answers">Answers: </label>
 <input id="answers" name="answers"
     type="radio" value="affirmative"
     onchange="record(this)">Yes</input>
 <input id="answers" name="answers"
     type="radio" value="negative"
     onchange="record(this)">No</input>
```

The JavaScript function "record" is responsible to instantiate the value of a special form control whose id is "task" and then submit the form. This control will contain the preprocessed observation data and will be further converted into HSL-ready notation by a PHP script on the web server.

Figure 7. JavaScript Function record()

```
function record(sender) {
  if (sender.type == "checkbox")
  {
  document.getElementById("task").value
     = sender.name;
  }
  else if (sender.className ==
           "property")
  {
  document.getElementById("task").value
     = sender.name + "/" + sender.value;
  }
  else
  {
  document.getElementById("task").value
     = sender.name + " " + sender.value;
  }
  document.getElementById("data").
             submit();
}
```

In general, the PHP program called by the form need only know the participants, the name of the task control and its value, and in certain cases, the object of the task. As only controls in the task fieldset are set with the onChange attribute, the subject of the action and its object (when relevant) must be set prior to setting the value of a task control. In this way, recording observations is linguistically similar to German grammar, where verbs appear at the end of sentences.

The PHP application records a running

observation log on the web server which includes timestamps for the observations. The log can easily be converted into compliant HSL notation.

Genericizing the data collection tool

In theory, the data collection tool used in the previous section could be built via an application generator. This generator would essentially allow for the same data collection application to be used for any observations. In some cases an observer may have a basic idea of what they expect to see. However, it is more likely that the observer will need to add new components to the application as different observations are being made.

In our original application, several workers and customers are hard coded into the list of participants. But what happens if more individuals enter the scene? A more generic version of the tool would include a button for adding participants that can be used as the observer sees a new potential participant. When the button is pressed a list of suggested new participants should appear to help minimize the need for typing inputs. For example, "worker 6" and "customer F" could be on the list of suggested participants at the deli counter. In the case that neither of these would be the best titles for the participant, a text field could exist to allow for a new title.

Other elements such as props could be added the same way, although it may be more difficult for the application to generate suggestions for these. Whereas participants in a linkage may be given generic titles (which may also represent their role parts in the case of customers and workers) props are more likely to be their own specific entity. That being said, users will most likely have to manually input information for new props. Although, frequently used props could be given as suggestions so the user does not have to recreate the same prop for every observation.

Tasks are the most difficult to genericize. Dynamically adding tasks is not as simple as the other elements which require only value inputs. Checkboxes, radio buttons, text fields and possible other sources for user input need to be considered for each task. It is feasible for the user to build tasks by selecting which option to have, however this may be a tedious task. Creating new tasks during the observation may be too difficult, and therefore the user should decide ahead of time the important tasks to include. As in the original deli counter application, the option for entering "other" tasks as text will still be necessary.

Outcomes and Future Work

A key outcome of our work was the actual use of the observation tool in the field to collect data for linkages similar to the one described in Figure 2.

Figure 7. Actual Output from the Observation Tool (corresponds to Figure 2)

```
2013-04-13 06:38:26  <LEDmonitor/43>
2013-04-13 06:38:43  [Worker 1]<calls 44>
2013-04-13 06:38:55  [Worker 1]<calls 45>
2013-04-13 06:39:12  <LEDmonitor/45>
2013-04-13 06:39:22  [Worker 1]<inquires>
2013-04-13 06:40:16  [Customer A]
                      <orders sliceable>
2013-04-13 06:40:33  [Worker 1]
                      <processes slices>
2013-04-13 06:41:27  [Worker 1]
                      <processes weights>
2013-04-13 06:41:54  [Worker 1]
                      <validates weight_ok>
2013-04-13 06:42:04  [Customer A]
                      <answers affirmative>
2013-04-13 06:42:15  [Worker 1]
                      <anything_else>
2013-04-13 06:42:29  [Customer A]
                      <answers negative>
2013-04-13 06:42:37  [Worker 1]
                      <delivers_closing>
```

By genercizing the data collection tool we can expand the observation application and create a framework for producing simple dynamic observation tools. This tool also has potential use outside of linguistics. Although terms such as participant, role part, prop and tasks are specific to HSL, their definitions and extensions are useful in other social and physical sciences.

Consider an anthropologist studying a tribal society in Africa. His or her observations will need to include participants and tasks as well. Although the term task may not actually be used by anthropologists, tasks can still represent the events taking place during the observation.

CONCLUSION

As Hard Science Linguistics was founded based on a systematic approach, it is well suited to the development of modeling and observation tools. In this article, we demonstrated an dynamic method, object-orientation approach to modeling HSL interactions and a technique to record field observations. Both tools can be transformed into more generic production applications. The modeling tool, specifically, can also be taken to the next level utilizing prototype-based languages which are becoming increasingly popular in computerized modeling projects.

REFERENCES

Aldrich, Jonathan, Sunshine Joshua, Saini, Darpan & Sparks, Zachary (2009). Typestate-oriented Programming. Procedings of the 24th ACM SIGPLAN conference on Object-Oriented Programming System Languages and Applications.

Burazer, Lara. 2006. Human Linguistics and Referring in the Real World. *LACUS Forum 32:155-161.*

Carlson, Brent, daCruz, Valquiria C., Graser, Tim, Marandici, Mircea P., Pietrocarlo, Gary J., Tost, Andre, Woods, Craig D., & Yelchur, Ravindran (2004). System and Method for Defining, Configuring and Using Dynamic, Persistent Java Classes. US Patent No. 6,766,324 B2.

Duan, Yucong, & Cruz, Christophe (2011). Formalizing Semantic of Natural Language through Conceptualization from Existence. Journal of Innovation, Management and Technology, Vol. 2, No. 1.

Graesser, A. C., Hu, X., & McNamara, D. S. (2005). Computerized learning environments that incorporate research in discourse psychology, cognitive science, and computational linguistics. In A.F. Healy (Ed.), *Experimental cognitive psychology and its applications: Festschrift in honor of Lyle Bourne, Walter Kintsch, and Thomas Landauer* (pp. 183-194). Washington, DC: American Psychological Association.

Layka, Vishal, Judd, Christopher M., Nusairat, Joseph F., & Shingler, Jim (2013). Beginnning Groovy, Grails and Griffon. Apress, New York, NY.

Richardson, Chris. (2009). ORM in Dynamic Languages. *Communications of the ACM, 52(4) pp. 48-55.*

Salzman, Lee & Aldrich, Jonathan (2005). Prototypes with Multiple Dispatch: An Expressive an Dynamic Object Model. *Proceedings of the 19th European conference on Object-Oriented Programming.*

Sypniewski, Bernard Paul (2001). Hard-Science Linguistics as a Formalism to Computerize Models of Communicative Behavior. In *International Computing and Information Technologies – Exploring Emerging Technologies,* ed. George Antoniou and Dorothy Deremer, World Scientfic, River Edge, NJ and Singapore.

Sypniewski, Bernard Paul. (2008). Snake in the Grass. *Language, Communication and Social Environment, 6.* Voronezh State University.

Sypniewski, Bernard. (2010a). Hard Science

Linguistics looks at Humor. 2010 Ars Grammatica conference, Minsk State Linguistic University, Minsk, Belarus.

Sypniewski, Bernard Paul. (2010b). Introduction to Hard Science Linguistics. *The World of Linguistics and Communications.* Institute for Applied Linguistics and Mass Communication, Tver University, Russia. < http://tverlingua.ru/archive/028/01_28.pdf>.

Sypniewski, Bernard Paul (in press). Computers and Linguistics. Presented at *2011 LACUS Conference Proceedings*, Toledo, OH.

Yngve, Victor H. (1996). From Grammar to Science: New Foundations for General Linguistics. Amsterdam, NL: John Benjamins Publishing Company.

Yngve, Victor H. (2004). Formalizing the Observer in Hard-Science Linguistics*.* In Yngve, Victor H. and Wąsik, Zdisław. *Hard Science Lingusitics.* Continuum, NY.