Generating Natural Language Aggregations Using a Propositional Representation of Sets

Susan Haller[1], **Barbara Di Eugenio**[2], **and Michael Trolio**[2]
[1]Computer Science Department
University of Wisconsin–Parkside
`haller@cs.uwp.edu`

[2] Computer Science Department
University of Illinois at Chicago
{`bdieugen,mtrolio`}`@cs.uic.edu`

Abstract

We present a method for aggregating information from an internal, machine representation and building a text structure that allows us to express aggregations in natural language. Features of the knowledge representation system, a semantic network, allow us to produce an initial aggregation based on domain information and the competing aggregate structures. In the £nal stages of realization, the network representation allows us to use low-level (below the level of a clause) information to perform linguistic aggregation. The test bed for this work is an interactive tutoring system on home heating systems.

Introduction

When speaking or writing, people display a remarkable ability to avoid duplicating information that they have already presented, or that is directly inferable from what they have said. In natural language generation systems, *aggregation* organizes text content according to common features and concepts to avoid redundancy in dialogue.

The machine-generated dialogue in Figure 1 motivates the need for aggregation. The output is produced by an interactive tutoring system (ITS) built using the VIVIDS/DIAG-based framework. The system tutors students about repairing a home heating system. This output is produced in response to a query about why the furnace igniter continues to ignite in the heating system's start up mode. The user interacts with the tutoring system by clicking on the graphic display corresponding to the £re door to the furnace. In its response, the ITS nominalizes this action as the "visual combustion check indicator".

One simple way to make the text more understandable is to order the sentences according to units that always, sometimes, or never produce the abnormality. This would group sentence 8 with 2-4. However, the redundancy in sentences 2-8 still makes the text dif£cult to understand. If spoken, the text becomes nearly incomprehensible. Good aggregation is critical in human dialogue systems.

There are several issues in regard to producing appropriate aggregations. One is choosing an ontology that facilitates aggregating features of the content. For example, the

```
1   The Visual combustion check is igniting
    which is abnormal in this startup mode
    (normal is combusting).
2   Oil Nozzle always produces this abnormal-
    ity when it fails.
3   Oil Supply valve always produces this ab-
    normality when it fails.
4   Oil Pump always produces this abnormality
    when it fails.
5   Oil Filter always produces this abnormal-
    ity when it fails.
6   System Control Module sometimes produces
    this abnormality when it fails.
7   Igniter assembly never produces this ab-
    normality when it fails.
8   Burner Motor always produces this abnor-
    mality when it fails.
```

Figure 1: A response from a VIVIDS/DIAG tutor in the home heating system domain

illustration in Figure 1 makes it clear that we could aggregate on the level of certainty with which a failed unit causes the abnormal visual combustion check. But what is not clear from the dialogue is that the units mentioned could be grouped according to the heating subsystem that they belong to: the oil burner or the furnace.

We can also aggregate the text based on features that are not domain related. For example, aggregation on the lexical level might be appropriate. In Figure 1, several of the units have names that are modi£ed by the word "oil" (by virtue of being part of the oil burner). This might compel us to aggregate these names into a single phrase – "oil nozzle, supply valve, pump, and £lter" if enough units fall into the same aggregate group. Another consideration is the branching factor of an aggregation. If we aggregate on the certainty with which a failed unit causes the abnormality, there are three attribute values all with units in them: always, sometimes, and never. However, there are only two aggregate values if we aggregate on heating subsystem: furnace and oil burner. The state of the discourse might also make one aggregation preferable to another. For example, if the conversational focus is on the furnace, it might be appropriate to present all information about the furnace at once. This dictates aggregating on subsystem. As (Reiter & Dale 1997) points out,

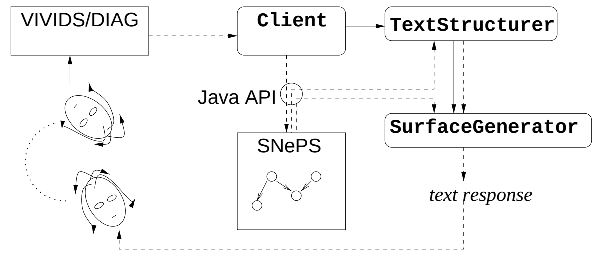

Figure 2: System Con£guration

one of the most dif£cult problems with aggregation is deciding on which among the numerous potential (and possibly con¤icting) aggregations to perform.

In the next section, we describe the relevant architecture of our system and pertinent features of the the knowledge representation system and generation tool that we have integrated. Then, we work through the example above to show what our system produces and why. We conclude with a discussion of our work to date and future work.

System Overview

DIAG is a shell to build ITSs that teach students to trouble shoot systems such as home heating and circuitry(Towne 1997a; 1997b). DIAG builds on the VIVIDS authoring environment, a tool for building interactive graphical models of complex systems(Munro 1994). A typical session with a DIAG application presents the student with a series of troubleshooting problems of increasing dif£culty. At any point, the student can consult the built-in tutor in one of several ways. For example, if the student is unsure what the reported status of a system indicator means, she can consult the tutor about it. Figure 1 is an example of the original DIAG system's output in a case like this. After selecting content, the original DIAG uses simple templates to assemble the text to present to the student. The result is that DIAG's feedback is repetitive. Furthermore, this problem is compounded by the fact that DIAG applications involve complex domains with many parts that DIAG might need to report on in a single response.

Figure 2 shows the basic con£guration of subsystems that we use to interface to a VIVIDS/DIAG tutor. Solid lines indicate control ¤ow, and dashed lines indicate communication and data ¤ow. A Java `Client` establishes a TCP/IP connection to the VIVIDS/DIAG tutor, listens to the tutor, and intercepts the content of each tutor response. The tutor's representation of the content is impoverished, so the `Client` interfaces to the Semantic Knowledge Representation and Reasoning System (SNePS) to represent the content of the tutor's response in a semantic network.

SNePS is a semantic network knowledge representation and reasoning system with a logic that targets natural language understanding and commonsense reasoning(Shapiro 2000). A SNePS network is said to be *propositional*, because all propositions in the network are represented by nodes. In SNePS, there are *nodes* and labeled, directed edges called *arcs*. All nodes represent concepts, either objects or propositional concepts. When information is added to the network, it is added as a node with arcs emanating from it to other nodes. A SNePS feature that aids linguistic aggregation is that each concept in the network is represented by a unique node. This is called the *Uniqueness Principle*.

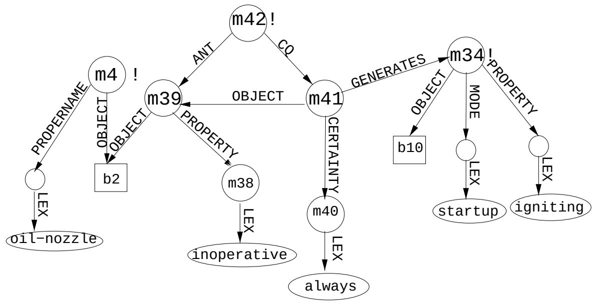

Figure 3: SNePS Network for: If `m39` [`b2` is inoperative], then `m41` [`m39` generates (causes) `m34!` (`b10` igniting in startup mode) with certainty `always`]

Figure 3 shows an example of SNePS network that is constructed to make part of the response in our example. SNePS-2.5 uses an assertion ¤ag (represented with a `!`) to distinguish what is asserted as true from other propositions. `m42!` is a rule that asserts that if `m39` (`ANT` = antecedent) then `m41` (`CQ` = consequent). `m39` is the proposition that object `b2` is inoperative, and `m41` is the proposition that the situation represented by `m39` always causes `m34!`. `m34!` asserts that object `b10` is igniting in startup mode. In summary, `m42!` asserts that if `b2` is inoperative, then its failure always causes `b10` to ignite in startup mode. Another network segment `m4!` asserts that `b2` is called a `oil-nozzle`. While `m4!`, `m42!` and `m34!` are asserted as true, propositions `m39` and `41` are not asserted, indicating that they are not known to be true. The semantic network is loaded apriori with static information about the home heating system, for example, that the oil-nozzle is a component of the oil burner. Then transitory information (information for the speci£c response like Figure 3) is added by the `Client`.

The `Client` invokes a Java `TextStructurer` to build a text structure of rhetorical relations based on *Rhetorical Structure Theory* (RST) (Mann & Thompson 1988). Each rhetorical relation has at least one *nucleus*, the core idea to be communicated. A nucleus can be augmented with a *satellite*, a clause that is related to the nucleus but is not required. According to RST, the communication of the satellite in addition to the nucleus also communicates to the listener the relationship between the satellite and the nucleus. Figure 4 shows a rhetorical relation built from the SNePS network in Figure 3. The nucleus is at the arrow head, `m41`, and the satellite is `m39`. The arrow going from `m39` to `m41` indicates that, as a satellite, `m39` is a condition for `m41`. The `TextStructurer` queries the SNePS network to build the text structure starting with a nucleus, in this case `m41`.

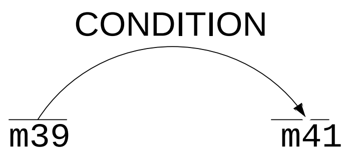

Figure 4: Text structure built using `m39` and `m41` in Figure 3

In this example, the `TextStructurer` applies a rule that states that:

```
Given a nucleus,
    if there is a node in the network that is
        NOT asserted and that
        is an antecedent to the nucleus
    then
        the node is a satellite to the nucleus
        in the CONDITION relationship
```

The actual query looks for a `ANT-CQ` arc relationship between an unasserted node in the network and nucleus `m41`. Node `m42!` expresses this relationship, and `m39` is added to the text structure as a satellite in a CONDITION rhetorical relationship. When this text structure is traversed by the surface generator, it can be used to determine a linear ordering of the information in these nodes and to add cue phrases to the text. For example, Figure 4 could be realized as "if `m39` then `m41`" or "`m41` whenever `m39`".

The `TextStructurer` makes repeated queries until no new text structure can be built. Then the `TextStructurer` invokes the `SurfaceGenerator` to realize the complete text structure in natural language. Since the nuclei and satellites in the text structure are SNePS nodes, the `SurfaceGenerator` also queries the network to build a linearization. The `TextStructurer` and the `SurfaceGenerator` are written using Exemplars, a object-oriented framework for building generation grammars that allow for mixing template-style processing with more sophisticated text planning (White & Caldwell 1998).

An Example

When the user asks the VIVIDS/DIAG tutor about the visual combustion check indicator, the `Client` receives a set of records. The initial record shows that the user wishes to consult the tutor about a heating system indicator, the visual combustion check indicator. The records that follow are about units that could be replaced (called "replaceable units" or RUs) and the certainty with which each RU causes the abnormal indication. After adding the information to the SNePS network, the `Client` invokes the `TextStructurer` with the values `"ConsultIndicator"` and `"visual combustion check"`.

The `Client` does not pass the `TextStructurer` any content for the response. The `TextStructurer` is passed the task to perform: building a response that consults the user on the visual combustion check indicator. As discussed, different aggregation issues become important at different stages of building the text structure. Therefore, we let the `TextStructurer` select content as it builds and re£nes the text structure.

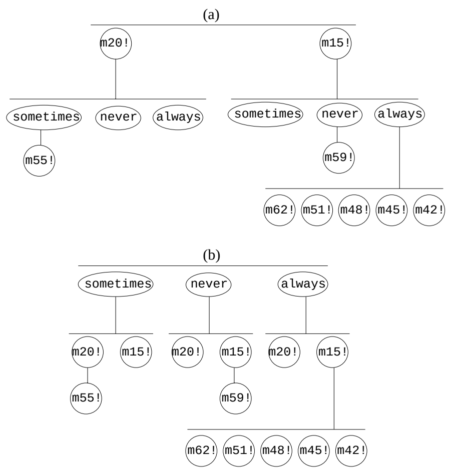

Figure 5: (a): `Aggregation` by system then certainty and (b): `Aggregation` by certainty then system

At the £rst stage, the `TextStructurer` queries the SNePS network for the initial state of the visual combustion check indicator and all assertions in the network about what could contribute to this state. This results in a query that returns nodes like `m42!` in Figure 3. We will refer to the set of nodes like `m42!` as the *certainty statements*. At this point in our work, the `TextStructurer` always aggregates certainty statements on two dimensions:[1] (1) the subsystem of the heating system that each RU belongs to and 2) the certainty with which an RU failure can cause the indicator abnormality. We refer to these as the *system* and *certainty* aggregation dimensions. The `TextStructurer` aggregates the certainty statements on (a) system, and within each system, on certainty. It also aggregates them on (b) certainty, and within each level of certainty, by system. Figure 5 shows the two `Aggregation` objects. Currently, we select the aggregation with the smaller initial branching factor. Since the system dimension has only two values, the `TextStructurer` selects aggregation (a)in Figure 5 to use as the initial nuclear content of the text structure.

Next, the `Aggregation` structure is mapped to appropriate rhetorical relations to make a text structure. The £rst aggregation dimension is by system, and it has two values: asserted nodes `m20!` and `m15!`. Space does not permit showing these nodes. However, `m20!` asserts that the RU mentioned in certainty statement `m55!` is a component of the furnace system, and `m15!` asserts that all the RUs mentioned in certainty statements `m59!`, `m62!`, `m51!`, `m48!`, `m45!`, and `m42!` are components of the oil burner. This division is based on an arbitrary (from a language point of view) division of the heating system into units. Therefore, no rhetorical relationship exists between `m20!` and

[1]However, we are not limited to two dimensions. We can aggregate over any number of dimensions.

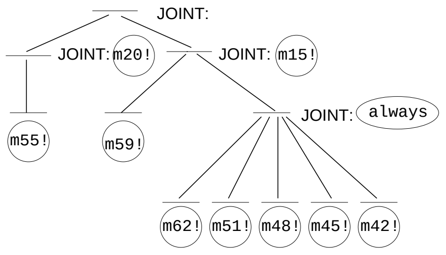

Figure 6: The text structure built from the initial aggregation

m15! and they are mapped to the only *non-relation* in RST: JOINT.

The nodes in the sub-aggregation under m15! are aggregated by scalar values "always", "sometimes", and "never". At this point in our work, this aggregation is also realized with a JOINT. The TextStructurer builds the initial text structure shown in Figure 6. Notice that each JOINT relation is associated with a node, for example,

Node m15! is an assertion about all the nodes in the JOINT: m59!, m62!, m51!, m48!, m45! and m42!. Namely, m15! asserts that an argument to each node in this group is a unit of the oil burner assembly.

The top-level JOINT is not associated with a node. This JOINT represents the initial aggregation, based on no represented knowledge, and only the branching factor of the aggregation. The only effect that the top-level JOINT relation will have is to make sure that the information under the JOINT labelled m20! and the information under the JOINT labelled m15! are presented as parallel text structures. Similarly, the JOINT labelled with m15! ensures that parallel text structures will be used for the content of node m59! and the third-tier JOINT labelled with the base node, always. Since this node is not an assertion (or least a propositional node), there is no way to express what the £ve certainty statements have in common in this JOINT, namely, that these are all assertions about units that always cause the abnormality being discussed.

Expanding the Structure

The TextStructurer expands the text structure in 6 with satellites. Figure 7 shows this expansion. In our example the top-level, multi-nuclear JOINT is augmented using a non-volitional result relation (NVRESULT) to add the satellite m34!. This will in turn be augmented with m36! as a satellite using an ELABORATION relation.

Structuring Within

The TextStructurer uses a second process to structure text that we call *structuring within*. Structuring within becomes necessary because the initial aggregation can contain clauses that are about other clauses. The SNePs logic is not strictly £rst-order. SNePS nodes can and often do represent

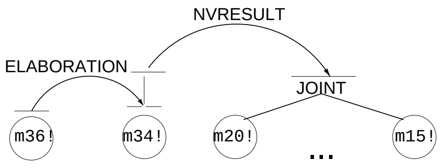

Figure 7: The text structure with satellites added

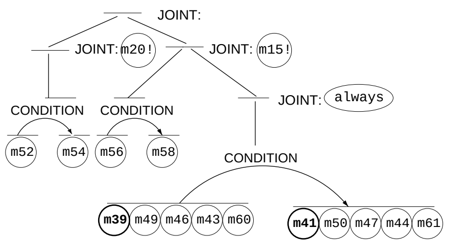

Figure 8: Figure 6 after structuring within

propositions about other propositions. So that we can aggregate over any content, we must be able to structure content internally that is already part of an overarching text structure. As an example of structuring within, we use node m42! in Figure 3. Structuring within will replace m42! with a text structure in which the antecedent of m42! (node m39) becomes the satellite to the consequent of m42! (node m41) in a CONDITION rhetorical relation. The new text structure is given in Figure 4. All the certainty statements can be structured within similarly, replacing each with a CONDITION rhetorical relation. As a result, the text structure in Figure 6 is replaced with Figure 8.

If the text structure in Figures 7 and 8 were realized using only templates, the output would be

1 A visual combustion check indicator is igniting in startup mode.
2 The visual combustion check indicator igniting in startup mode is abnormal.
3 *Within the furnace system,* this is sometimes caused when the system control module is inoperative.
4 *Within the oil burner*, this is never caused when an igniter-assembly replaceable-unit is inoperative.
5 This is always caused when a burner motor is inoperative.
6 This is always caused when an oil filter replaceable unit is inoperative.
7 This is always caused when an oil pump replaceable unit is inoperative.
8 This is always caused when an oil supply valve replaceable unit is inoperative.
9 This is always caused when an oil nozzle replaceable unit is inoperative.

The italicized phrases are the contribution of the assertions represented by nodes m20! and m15!. The repeated use of "this" is a reference to node m34! which represents the visual combustion check indicator igniting in startup

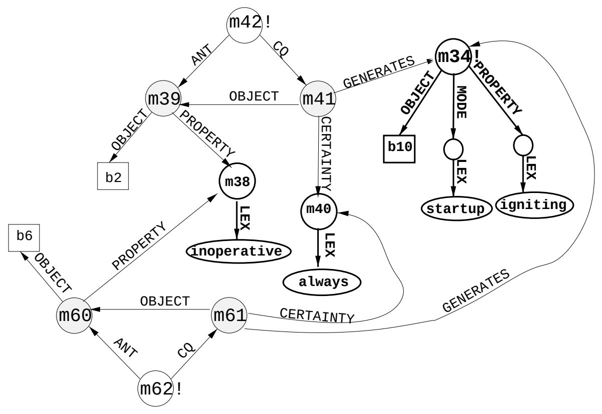

Figure 9: Structures shared in certainty statements `m42!` and `m62!`

mode.

Exploiting Structure Sharing

Domain-level information (as represented by propositions) cannot address the redundancy when the third-tier JOINT (lines 5–9) is realized. At this point, the basis for aggregation is below the clause level. This problem exists at some point in any ontology that is used.

We address the problem by exploiting the set logic of SNePS to perform aggregations on the basis of shared network structure at a representational level below the clause. Figure 9 shows two of the £ve certainty statements in the original third-tier JOINT (nodes `m42!` and `m62!`) and the antecedent and consequent of each of them (nodes `m39`, `m41`, `m60`, and `m61` – shaded). The two antecedents, `m39` and `m60`, share node `m38` in the same relationship (the `PROPERTY` relationship). Node `m38` represents the lexicalization of "inoperative". The two consequents `m41` and `m61` share node `m40` and `m34!` in the same relationships, as a certainty and a generated cause respectively. `m40` represents the lexicalization of "always". `m34!` represents that the visual combustion check is igniting in startup mode. All the antecedents in Figure 8 share `m38`, and all the consequents share `m34!` and `m40`.

During surface realization, network queries return shared nodes when all the antecedent nodes are used as a set argument in the query. Nodes like `m38` are returned. Similarly, we query the network for information about the set of consequents and return `m40` and `m34!`. These shared structures are realized once. Hence `m34!`, one of the two nodes shared by the consequents in the generates relation is expressed as "this is .... caused". Together with the other shared node, `m40`, the consequents are collectively realized as "this is always caused". Similarly, the node shared among the antecedents, `m38`, is expressed once as "... is inoperative". Since the unit names do not share structure, they are enumerated. As a result of structure sharing, we generate the the following aggregation in place of lines 5-9 above.

```
5  This is always caused when a burner
   motor, oil filter, oil pump, oil sup-
   ply valve, or oil nozzle replaceable
   unit is inoperative.
```

Discussion and Future Work

Semantic networks have capabilities similar to a relational database. In addition, they have great representational power. We have exploited these features to generate aggregations in natural language. As our example illustrates, the network allows us to aggregate based on domain information, for example, system and certainty. We can also aggregate based on domain-independent features. The antecedent-consequent relation is an example of this. Furthermore, during surface realization, shared structure is used to aggregate the text below the clause level.

We are currently expanding the system in preparation for testing and evaluation with human subjects. In future work, we want to automatically determine aggregation dimensions from a set of nodes that are to be aggregated. We will also consider the state of the discourse in choosing and ordering aggregation dimensions.

References

Mann, W. C., and Thompson, S. A. 1988. Rhetorical structure theory: Towards a functional theory of text organization. *TEXT* 8(3):243–281.

Munro, A. 1994. Authoring interactive graphical models. In Jong, T.; Towne, D. M.; and Spada, H., eds., *The Use of Computer Models for Explication, Analysis, and Experiential Learning*. Springer Verlag.

Reiter, E., and Dale, R. 1997. Building applied natural language generation systems. *Natural Language Engineering* 3.

Shapiro, S. C. 2000. SNePS: A logic for natural language understanding and commonsense reasoning. In Iwanska, L. M., and Shapiro, S. C., eds., *Natural Language Processing and Knowledge Representation*. AAAI Press/MIT Press.

Towne, D. M. 1997a. Approximate reasoning techniques for intelligent diagnostic instruction. *International Journal of Arti£cial Intelligence in Education*.

Towne, D. M. 1997b. Intelligent diagnostic tutoring using qualitative symptom information. In *AAAI Fall Symposium on ITS Authoring Tools*.

White, M., and Caldwell, T. 1998. Exemplars: A practical, extensible framework for dynamic text generation. In *Proceedings of the Ninth International Workshop on Natural Language Generation*.