HANDLING EXCEPTIONAL CONDITIONS IN PSN

Yves Lesperance

Department of Computer Science
University of Toronto
Toronto, Canada
M5S 1A7

ABSTRACT

This paper describes a scheme for handling both exceptional objects and classes and exceptional conditions that arise in the execution of programs, within a knowledge representation formalism. The scheme consists of two mechanisms: the excuse, which allows the justification of specified constraint violations in instances of a class through membership in a second class within designated contexts, and the mapping, which permits the specification of similarity relationships between the definitions of two objects, so that arbitrary elements of these definitions may be copied or inherited (a flexible IS-A). Exceptions in programs are handled through an extension of the excuse mechanism.

1.0 INTRODUCTION

In order to perform intelligently , a system must possess a model of its world and be able to use it to deal with the often unexpected situations that arise. The knowledge in this model (knowledge base) is organised in terms of a system of categories. The cathegories may be explicit, as in frame systems [Minsky 74], or more implicit as in logical formalisms. Exceptions in representation systems arise as a result of (1) the sometimes unpredictable nature of the world, which produces atypical situations, and (2) the inadequacies of current representation formalisms in dealing with "natural" concepts (as used by people). These exceptions manifest themselves through the violation of some constraint during the lifetime of the knowledge base.

A simple classification of exceptional conditions will help in finding ways to deal with them. Generic exceptions can first be distinguished fron individual exceptions, as the former pertains to constraints violated in the definition of a category rather than in particular individual objects. Individual exceptions can be further subdivided into static exceptions, which arise while the systems is attempting to instantiate or recognize an object (basic operations at the top-level), and dynamic exceptions, which are encountered during the execution of a user defined program.

This paper sumarizes an exception handling system develloped for the PSN representation formalism [Levesque 79], which is explained in details in [Lesperance 80]. The seminal ideas for the system came from [Minsky 74], where two ways of recovering from failure in a frame system are suggested. First, it may try to create an excuse for the exceptional condition with an appropriate reason. In this approach, the failure is seen as arising from the fact that the defective object is really an instance of two frames which interact, thus the object does not satisfy perfectly the ideal defined in one of the frames. The knowledge necessary to make the repair should be attached to a higher thematic context frame. The second approach involves using the local advice embedded in a similarity network to replace the defective frame by a more appropriate one.

The two approaches reflect the distinction between individual and generic exceptions. In the first case, we do not wish to create new categories for every single exception, thus an excuse mechanism has been devised to allow the handling of both static and dynamic exceptions and the maintenance of the consistency of the knowledge base. The excuse mechanism has been influenced extensively by exception handling mechanisms develloped for programming languages, [Levin 77] in particular. These mechanisms allow the mainline of the program to be expressed without cluttering it with the code required to handle exceptional conditions. Moreover, the handling code for the condition is attached to the caller or user of the program module which raised the exception, allowing for a context dependent recovery from the exception. This facility permits the use of a procedure even if the conditions for which it was designed are not satisfied, as long as the exceptions that will be raised can be handled by its caller or user. For generic exceptions, the problem lies in the insertion of the category into the existing hierarchies, especialy when the inheritance of only part of the definition of the category is desired. This has been done through a mapping mechanism inspired from [Moore 73], which makes explicit the inheritance process of definition elements and gives control to the user over it when this is needed.

The developpement of this system is seen as a step in the direction of improved flexibility for

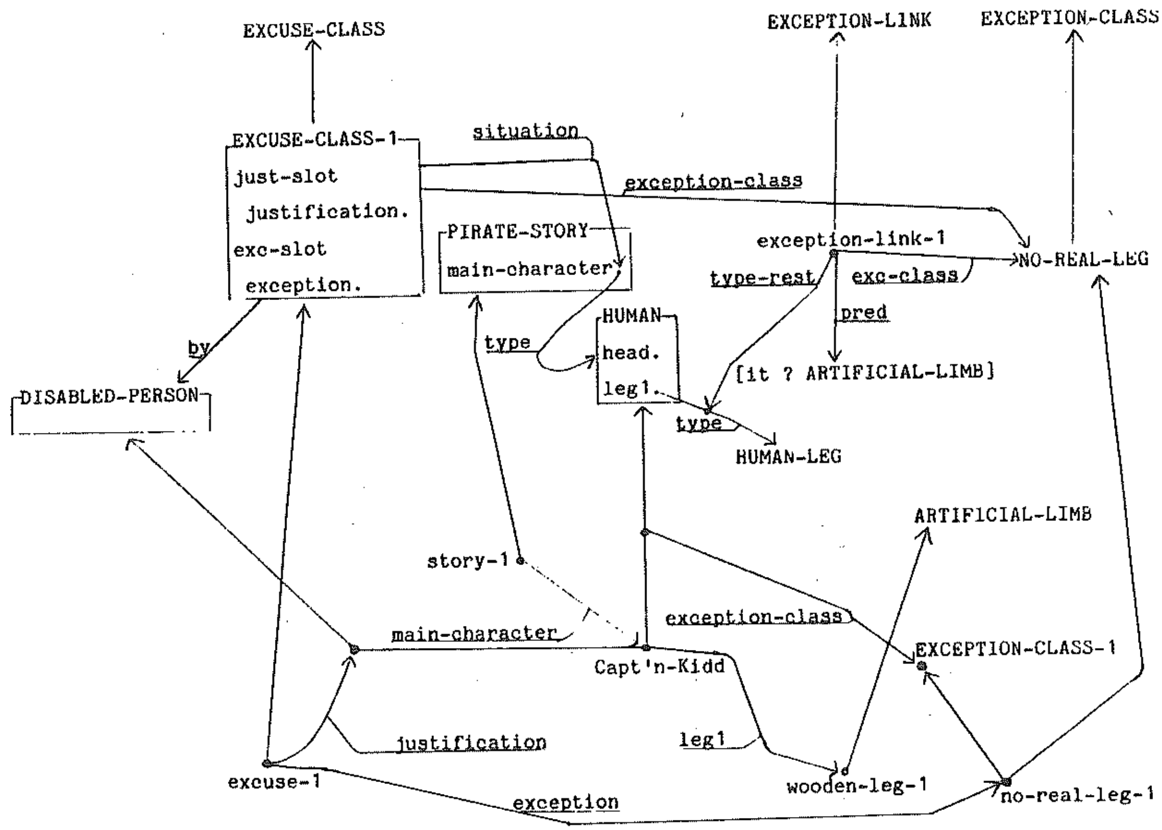

Figure 1 - Example of excuse for static exception.

representation formalisms, both for practical purposes and modelling adequacy. The system can be readily adapted to most other semantic network or frame based formalisms. The approach taken emphasizes the knowledge base definition aspect, but generality has been preserved. Before the system can be explained, an overview of its host formalism must be given.

2.0 OVERVIEW OF PSN

The PSN formalism grew out of a desire to develop a facility for defining semantic network knowledge bases with well defined semantics. The formalism is basicaly procedural, as the semantics of classes, which represent generic objects, are defined in terms of four attached programs, which prescribe the behavior of the class under the operations of instantiation, removal of an instance, testing for membership and fetching of all instances. Classes are represented graphicaly by their external name in capitals, for example "HUMAN" or "EXCEPTION-CLASS" in figure 1. Whenever an individual object is made an instance of a class, the appropriate attached program is executed, this allowing the desired inferences (antecedent theorems) to be added to the knowledge base. Similar action is taken in the case of the three other operations. Simple token objects are represented in the graphic notation by their external name in lower case, for example "Capt'n-Kidd" in figure 1. The INSTANCE assertion is represented by an unlabled single line arrow. Incidental relationships between objects (the links in traditional semantic networks) are represented by a class of objects called relations, whose semantics are also defined by four programs. The instances of relations are assertions of the relationship between two specific objects.

This basic procedural PSN is augmented with declarative facilities which help in the organization of the knowledge base. The defining properties of a class are grouped together to form the structure of the class, which consists of a set of slots which can have a type, restrictions, default, etc.. The structure of a class is represented by a box under the name of the class, for example "HUMAN" in figure 1, and slots by

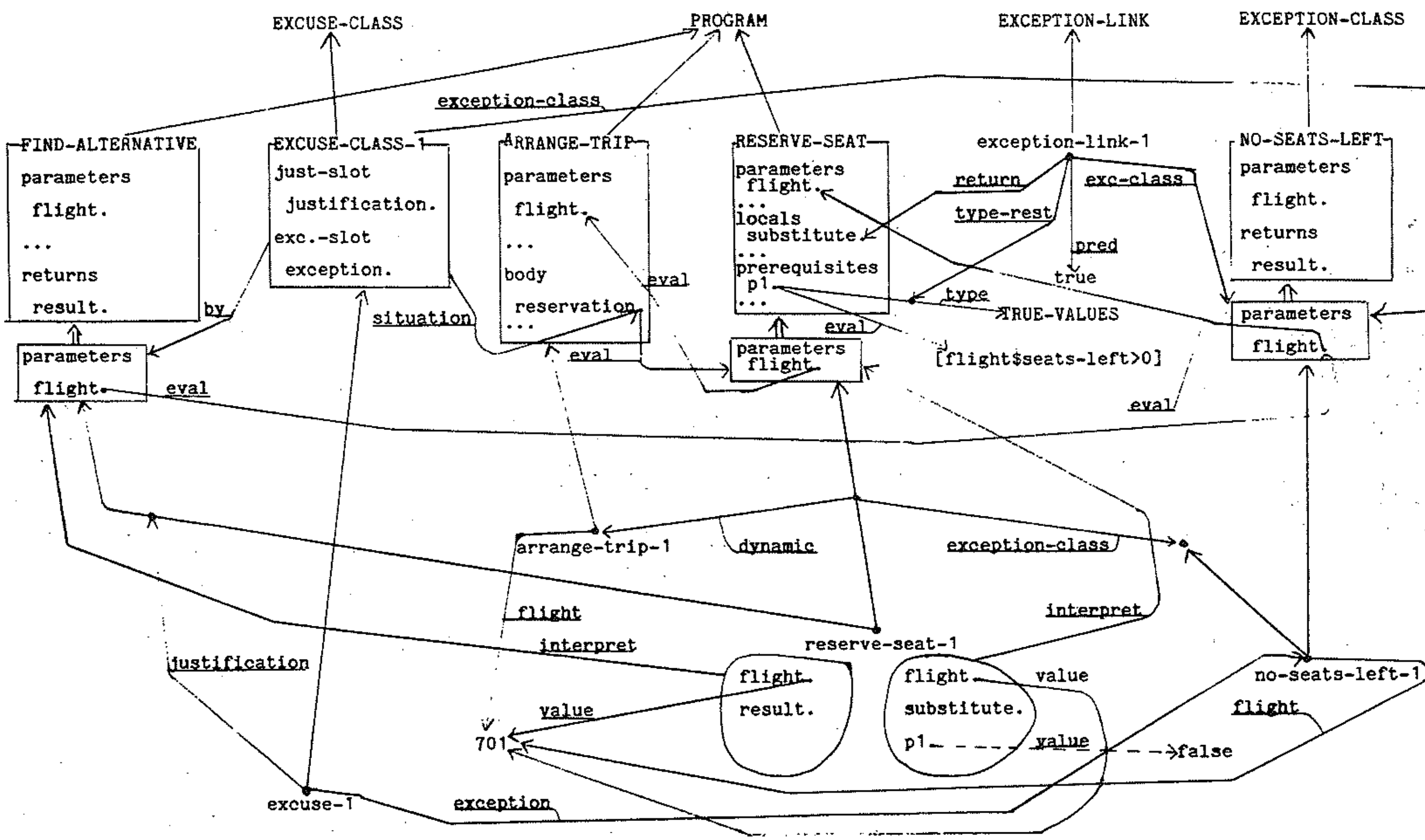

Figure 2 - Example of excuse for a dynamic exception.

their name with a node written in the box, for example "leg1". These slots can then be filled with values when an instance of the class has been created. This is represented by a link with the name of the slot as for the "leg1" of "Capt'n-Kidd" is "wooden-leg-1" in figure 1. The closure of these structural property value relationships forms the PART-OF hierarchy. The classes can also be organized in an IS-A or specialization hierarchy (represented by unlabled double line arrows, see figure 2). This facilitates the definition of the subclasses as the structure of the superclass is inherited by them. The slots can be refined but are required to satisfy the IS-A constraints, which guarantee that the subclasses are effectively specializations.
Slot values, in particular the four programs defining the semantics of classes, can also be inherited if necessary.

The instance hierarchy is not restricted to two levels and classes can be instances of metaclasses. This is used extensively in the definition of the formalism itself and many aspects of its behavior arise as a result of the definition of the metaclasses: CLASS, RELATION, OBJECT, PROGRAM,etc.. A metaclass can constrain the structure of its instances through its metastructure [Kramer 80], as the slots of the instance must be instances of the metaslots in the metastructure. Programs are represented as classes in the formalism, and thus benefit from all the declarative facilities. In figure 2, the program "ARRANGE-TRIP" calls another program "RESERVE-SEAT". Metaslots have been used to partition the slots into different categories: parameters, locals, etc.. To specify the desired parameter bindings and evaluations, a form is used (the box with no heading under "RESERVE-SEAT"). The programs are executed by creating processes which are instances of the programs, "arrange-trip-1" and "reserve-seat-1" in the example. The formalism also provides a context mechanism [Schneider 78, Schneider 80]. An object which is visible in a context is called a view. Context are used to implement inheritance, structures being essentialy special forms of contexts. A slot is inherited because it is visible (a view) in the structure of subclasses.

The only differences with some previous versions of PSN are the use of valuers to implement manifestations (ex: John as a taxpayer) as in [Schneider 78], which are needed for the proper treatment of dynamic exceptions, and the ability to refer to most systems assertions (INSTANCE, type, etc.). This feature can be simulated without any extension to PSN by replacing the single link assertion reference by a triple link reference to the relation and its arguments.

3.0 EXCUSES

3.1 STATIC EXCEPTIONS

The excuse mechanism takes care of objects which are instances of a class while violating some of the constraints associated to its slots. The exceptions which are raised by these violations must be handled by the class of the object which has the defective object as one of its parts (slot value), thus one level up on the PART-OF hierarchy. This provides a basic form of context sensitivity to the mechanism. The handler attached to the "situation" is restricted to being a class of which the defective object must also be an instance, thus retaining Minsky's idea of frame interaction in a context.

Let's explore the mechanism in more detail by considering an example of static exception handling represented graphicaly in figure 1. Here, we have an object "Capt'n-Kidd", which would be a legal instance of the class "HUMAN", except for the fact that the value of its slot "leg-1", "wooden-leg-1", violates the type constraint of the "leg-1" slot definition in the class "HUMAN". The violation is precisely that "wooden-leg-1" is not an instance of "HUMAN-LEG". To characterize this type of constraint violation, an exception-class called "NO-REAL-LEG" is created. Then this class is associated to the type of the slot "leg-1" using an exception-link. When the system, attempting to fill the value of "leg-1" for "Capt'n-Kidd" will detect the type violation, it will find the exception-link and then, if the predicate of the link is satisfied, it will create an instance of the exception class "NO-REAL-LEG". The exception "no-real-leg-1" is attached to the INSTANCE link between "Capt'n-Kidd" and "HUMAN", which thus becomes an EXCEPTIONAL-INSTANCE link. This is done by making the exception an instance of an exception-class created especialy for the link. Many exceptions could be raised on the instance in the same way.

The rest of the mechanism concerns the handling of the exception where the system tries to build an excuse for the exception. For that, it climbs up one level in the PART-OF hierarchy and looks at the corresponding class to find an excuse-class. In the example, this corresponds to following the "main-character" assertion to "story-1", then looking at its class "PIRATE-STORY" and then finding "EXCUSE-CLASS-1". This excuse-class must have been attached to the slot whose value is the exceptional instance. For the excuse-class to be usable, it must be associated to the exception-class of which the exception is an instance. If this is the case, then the system tries to make the exceptional object an instance of the class which is the value of its "by" slot, which is "DISABLED-PERSON" in this case. Any desired checking for evidence for this type of excuse can be done at this stage. If the instantiation has been succesful, then an excuse is created, which associates the justification to the exception. In the example, this is "excuse-1". The excuse marks the

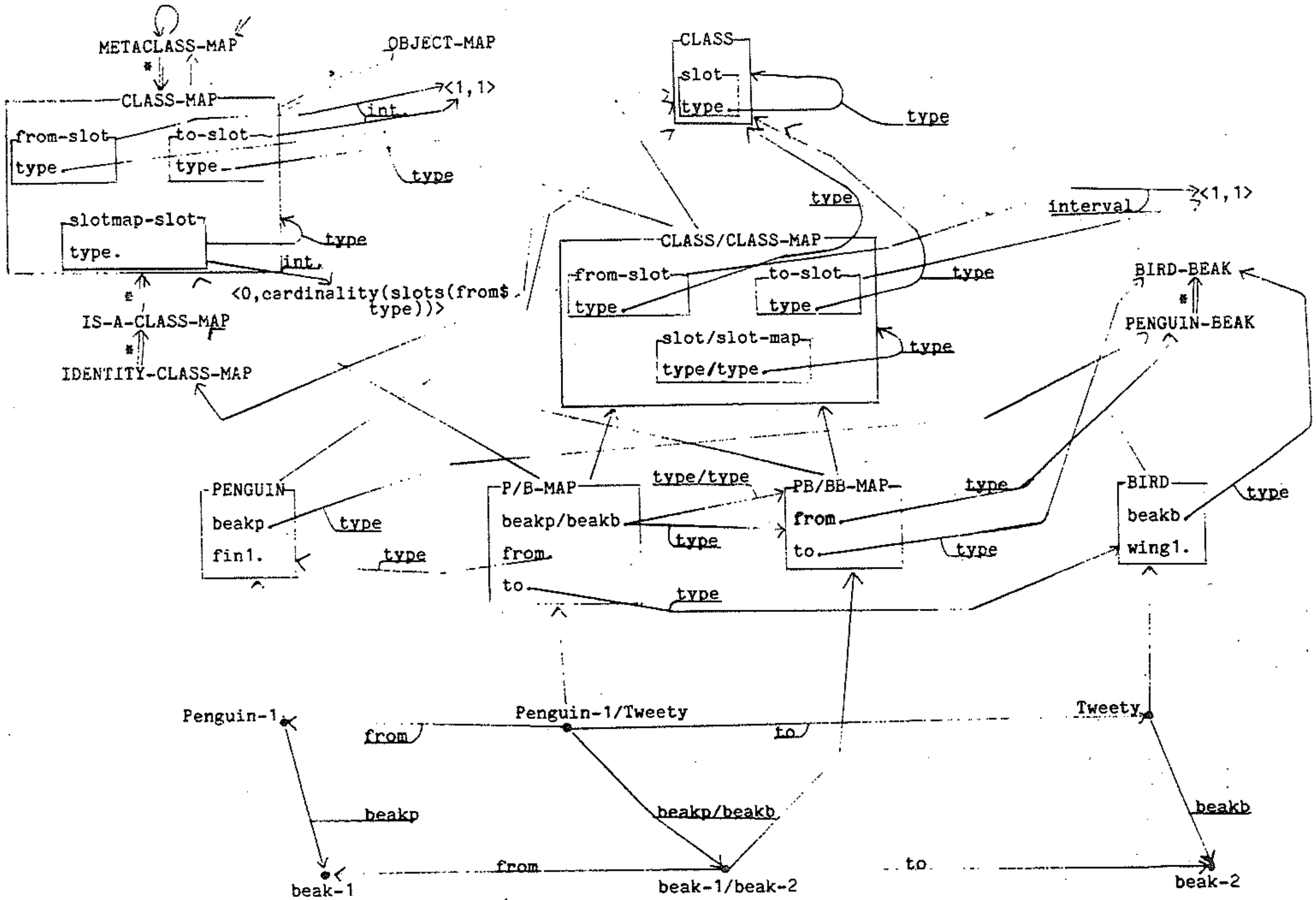

Figure 3 - Example of mapping.

succesful handling of the exception. If all the exceptions attached to an exceptional-instance link via its exception-class have been excused, then the link becomes an EXCUSED-INSTANCE link.

Exception-classes in this system have a two-fold function: they are abstract descriptions of the violations that arise and they allow an economical interface between the excuse-classes, which handle the violations, and the violations themselves, assuming that some violations will be treated in the same way. The use of the PART-OF hierarchy as a kind of context mechanism for exceptions is new to PSN, but resembles that of NETL [Fahlman 79]. The excuse mechanism also works nicely for cases of non-existant slot values. In this case, the special object "nothing" is given as a value. This can be treated as a type violation and be handled in the normal way.

3.2 DYNAMIC EXCEPTIONS

The excuse mechanism can be used to handle dynamic exceptions with a few extensions. It is natural to see exception-classes as the interface between the program context raising the exception and the one which will be selected to handle it. As these two belong to different levels of abstraction, it is necessary to provide parameter passing facilities with exceptions. These are defined as slots in the exception-class. The raising of an exception is similar to a procedure call, with the difference that the actual procedure to be invoked has to be selected by the system using the information provided by the excuse-classes. The scheme chosen requires the exception handling program to return control to the raiser of the exception after it has completed, as in [Levin 77]. This requires the definition of a returns slot in the exception-class.

In the example represented graphicaly in figure 2, a type violation has occured in the process "reserve-seat-1", which was invoked by "arrange-trip-1". The violation is on the prerequisite slot "p1", which checks whether some seats are available on the flight. As the value returned was "false", an instance of the exception-class "NO-SEATS-LEFT" is created ("no-seats-left-1") and attached to the INSTANCE link of the process. In the case of dynamic exception handling, the exception-link does not point directly to the exception-class, but to a form which is a subclass of it, allowing the parameter bindings to be indicated by "eval" assertions. A more important difference is the presence of a return slot value indicating which slot of the raiser should recieve the result of the evaluation of the exception handler.

After the creation of the exception, the system looks for an excuse-class (having the appropriate exception-class) attached to the slot that was being evaluated in the caller of the process that raised the exception. The dynamic hierarchy is used instead of PART-OF as it fills a similar role in dynamic objects like programs to that of part-of in static objects. Thus the "dynamic" assertion is followed from "reserve-seat-1" to "arrange-trip-1", where the "EXCUSE-CLASS-1" is located, from the "reservation" slot that was being evaluated. Then, the form which is the value of the "by" slot and a subclass of the "FIND-ALTERNATIVE" program is instantiated (executed), as the exception handler. Here again, a form is used to allow for the binding of parameters. The instance of the "by" class "FIND-ALTERNATIVE", is a manifestation of the same object "reserve-seat-1" that raised the exception. The explicit representation of the valuers (the ovals containing the value assignements to the slots) makes the separation of the two manifestations clear. The exception handling process thus appears as a tailoring of the process "reserve-seat-1" to fit the particular situation at hand. Once the instantiation has completed, an excuse is created ("excuse-1") for the succesfuly handled exception. Then, the "result" of the handler, that is the value of its slot which is an instance of the "returns" metaslot, can be passed back to the exception and to the process which raised it. This amounts in this case to set the local slot "substitute" to this value. Then, the process resumes after the point of interuption. A process can trigger an exception voluntarily by returning the special value "fail" in the same way as "nothing" in the static case.

3.3 INTERACTIONS WITH THE HIERARCHIES AND SEMANTICS

The immediate father in the PART-OF (dynamic) hierarchy is not always the best class to provide an excuse for an exception, but the scheme requires the exception to be reformulated in terms of the father class before it can be passed up higher, so as to preserve the abstraction structure. This is done in the static case by considering the unexcused exceptional object as violating the type of the father. In the dynamic case, the handler ("by" class) can also raise a new exception of its own, as it is treated as a part of the caller's context.

Even if it does not appears so by the examples given, it is intended that exception-links and excuse-classes be inherited with the slot they are attached to down the is-a hierarchy. They can also be refined and have to satisfy the is-a constraints (that their parts be identical or is-a, including the exception-class and the "by" class). This can be enforced by the formalism if these objects are defined as classes with slots representing the links, as in [Kramer 80]. However this solution is not totaly satisfactory. A default exception-class called "GENERAL-EXCEPTION-CLASS" is provided by the formalism to every slot defined, through the inheritance mechanism.

The excuse mechanism can be considered to be simply a syntactic extension of the original PSN formalism. The attachement of an exception-link and exception-class to a slot can be seen as the creation of a class which only differs from the original class by the required presence of the violation which would raise the exception. The attachement of an excuse-class to a slot effects a modification of its type, generalizing it to include some of these "violation" classes.

4.0 MAPPINGS

Our goal in designing the mapping mechanism was to define a very general construct which would (1) provide a facility for describing similarities that exist between objects and (2) allow the definition of classes in terms of other classes, including the copying of parts of their structure on a piecemeal basis to enhance expressive efficiency. The motivation for this came mainly from the lack of flexibility of the current IS-A construct, which is heavily felt whe dealing with natural concepts. In fact, IS-A should appear as a particular specialization of the general mapping construct and as such, it cannot be used in its definition.

An example of application of this more general mapping construct would be defining the class "PENGUIN" in term of the class "BIRD" by specifying a mapping from "PENGUIN" to "BIRD" which includes, as a submapping, saying that the "beak" slot of "PENGUIN" has a type which is a particular specialization of that of the "beak" of "BIRD". This is represented graphicaly in figure 3, where "Pb-MAP" is such a mapping (more details later). In this definition process, the user creates a mapping and expects the mapping instantiation program to create all objects and views not already existing and have them form the class being defined in terms of the other, as a side-effect of the mapping instantiation. Two aspects of the definition of mappings can thus be identified: their structure, which is concerned with the description of the relationship between the two objects, and their side-effects, which include object creation and manipulation of the structure hierarchy (contexts) to effect inheritance. The rest of the presentation concerns mainly the structural aspect as the other still needs to be worked out in details.

The main influences on the mapping mechanism have been the mappings of MERLIN [Moore 73], where the recursive aspect of their definition is taken, the "cables" of KLONE [Brachman 79], for the idea of structured inheritance, and the similarity networks of [Winston 75].

The main idea on which the mechanism is based is that any mapping of an object must also involve the mapping of its type(s), as it is an essential part of its definition. This requirement causes the structure of mappings to mirrors closely that of the INSTANCE hierarchy. If we return to our example in figure 3, the mapping "P/B-MAP" between the clases "PENGUIN" and "BIRD" is also a class and an instance of "CLASS-MAP". It contains a slot-mapping slot, "beakp/beakb", from the "beakp" slot of "PENGUIN" to the "beakb" of "BIRD". The type of this slot, "PB/BB-MAP", is another mapping class from the type of "beakp", "PENGUIN-BEAK", to the type of "beakb", "BIRD-BEAK". "PB/BB-MAP" would itself be expanded in the same way to map the slots of both classes. Now at the token level, there is an instance of "P/B-MAP", mapping "penguin-1" to "Tweety". It has as slot value a mapping between both "beak" slot values, which is an instance of "PB/BB-MAP". Thus, the mapping at the class level allows us to map the instances of the class. The structure of the mappings is exactly parallel to that of the classes mapped.

However, to satisfy completely our requirement, the types of the classes "PENGUIN" and "BIRD" must also be mapped. This is accomplished by "CLASS/CLASS-MAP", which maps the class "CLASS" into itself. Note that both "P/B-MAP" and "PB/BB-MAP" are also instances of this metaclass. The type of "CLASS" itself, "METACLASS", would also need to be mapped, but eventualy this will stop as "METACLASS" is only an instance of itself.

The classes that define mappings ("CLASS-MAP", "METACLASS-MAP", etc.) also allow us to create a taxonomy of mappings and differentiate between identity mappings, IS-A mappings and general similarity mappings. This is done by gradualy adding more constraints on the structure of mappings (e.g. the "interval" of "CLASS-MAP"), mainly on the metaslot controling slot mappings ("slot-map-slot"). This produces a pseudo-IS-A hierarchy of mappings. In the example, the "PB/BB-MAP" is an instance of "IS-A-CLASS-MAP" and its argument classes would satisfy the IS-A constraints. "CLASS/CLASS-MAP" is an instance of "IDENTITY-CLASS-MAP" as it maps a class to itself.

The mapping construct allows the representation of similarities of similarities, as mappings are simply objects like everithing else. It is also a powerful tool to study relationships involving the parts of objects as well as the objects themselves. An interesting question raised by the characterization of IS-A as a class of mappings is whether its set-inclusion aspect (instances of subclasses are instances of superclasses) is simply a side-effect of the IS-A constraints or a supplementary relationship. A mapping class can also be devised which exibits the constraints of the INSTANCE relationship. However, this abstract comparison of existing structures should not be confused with the INSTANCE assertion itself, which is the result of an external recognition process starting from sensory features and whose existence is assumed by the mapping mechanism.

5.0 COMPARISON TO OTHER SCHEMES

The only other representation formalism to give significant attention to the static and generic exception problems is NETL [Fahlman 79]. Its solution is much simpler than ours, being based on the insertion of "CANCEL" links in the virtual copy hierarchy to cancel inheritance when needed. This may be considered analogous to a mapping mechanism based on differences. There is no need for excuses as NETL neither does include a separate instance hierarchy nor programs. The mechanism is defined at a lower level of abstraction than ours (the user is concerned with the inheritance process) and is affected by the emphasis on retrieval. It does not offer the descriptive facilities of our solution and does not enforce any consistency or justification requirement.

The excuse mechanism for dynamic exception handling has many points in common with those of [Kramer 80] and [Mylopoulos 79]. However, it differs essentialy with that of [Kramer 80] on the question of where control should be returned after the completion of the exception handler. We require the resumption of the process which raised the exception, rather than return control to its caller. This makes it easier to ensure that the model is not left in an inconsistent state, is more efficient and promotes a more natural view of abstractions.

A more logical approach to exceptions has recently been proposed. Exceptions are seen as entities for which some default inference rule does not hold [Reiter 78](e.g. birds fly unless we can prove otherwise, for penguins the rule does not hold). Systems based on this principle maintain justifications for their assertions and reevaluate them as new facts are learned, which may contradict existing defaults deductions [Doyle 79]. If a satisfactory (non-monotonic) logic can be found to characterize these systems, it could improve greatly our understanding of the nature of exceptions and how to deal with them.

6.0 CONCLUSION

Some work remains to be done to achieve the full potential of the excuse mechanism. It should be possible to extend it so as to accomodate "structural" exceptions that arise on objects shared among many program contexts, which need to be propagated along the user hierarchy instead of the dynamic hierarchy [Levin 77]. This would involve a better integration of static and dynamic exception handling. The side-effects aspect of the mapping mechanism also need to be worked out in details.

It is certainly necessary to experiment with both mechanisms on a larger scale, to see whether they are really useful and suggest improvements. This would show in particular whether the whole-to-part style of object definition (where the object is created before its parts), which is necessary to take full advantage of the excuse mechanism, is practical.

REFERENCES

Brachman, R.J. (1979). "On the Epistemological Status of Semantic Networks",in Associative Networks: Representation and use of knowledge by computers, Findler, N.V. (Ed.), Academic Press, New York.

Doyle, J. (1979). "A Glimpse of Truth Maintenance", in Artificial Intelligence: An MIT Perspective, Winston, P.H. and Brown, R.H. (Eds.), MIT Press, Cambridge, Mass..

Fahlman, S.E. (1979). NETL: A System for Representing and Using Real-world Knowledge, MIT Press, Cambridge, Mass..

Kramer, B.M. (1980). "Representing Programs in PSN". Proc. 3rd Nat. CSCSI Conf., Victoria.

Lesperance, Y. (1980). Handling Exceptions in PSN. M.Sc. thesis, Dept. of Computer Science, Univ. of Toronto, to appear.

Levesque, H.J. and Mylopoulos, J. (1979). "A Procedural Semantics for Semantic Networks", in Associative Networks: Representation and use of knowledge by computers, Findler,N.V. (Ed.), Academic Press, New York.

Levin, R. (1977). Program structures for Exceptional Condition Handling. Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburg.

Mylopoulos, J., Bernstein, P. and Wong, H. (1979). A Language Facility for Designing Database-Intensive Applications. CSRG-TR-105, Dept. of Computer Science, Univ. of Toronto, to appear in TODS.

Minsky, M. (1974). A Framework for Representing Knowledge. A.I. Memo No. 306, MIT A.I. Lab., Cambridge, Mass..

Moore, J. and Newell, A. (1973). "How Can Merlin Understand ?", in Knowledge and Cognition, Gregg, L. (Ed.), Lawrence Erlbaum, Potomac, Md..

Reiter, R. (1978). "On Reasoning by Default", Proc. TINLAP-2, Urbana, Ill..

Schneider, P.F. (1978). Organization of Knowledge in a Procedural Semantic Network Formalism. Tech. Report No. 115, Dept. of Computer Science, Univ. of Toronto.

Schneider, P.F. (1980). "Contexts in PSN". Proc. 3rd Nat. CSCSI Conf., Victoria.

Winston, P.H. (1975). "Learning Structural Descriptions from Examples", in The Psychology of Computer Vision, Winston, P.H. (Ed.), McGraw Hill, New York.