N E W M E X I C O S T A T E U N I V E R S I T Y
C O M P U T E R S C I E N C E D E P A R T M E N T

Representation of Procedural Knowledge

Roger T Hartley

Abstract

A case is made for correcting a perceived imbalance between the usual ways of representing declarative and procedural knowledge. A proposed solution joins the traditional conceptual graph notation and elements of visual programming languages. The aim is to maintain the visual nature of both notations without greatly changing each one.

Representation of Procedural Knowledge

1. Introduction

We start with the premise that there is a clear duality between declarative and procedural knowledge [Ryle, 1949]. One form of knowledge is inadequate without the other, and thus choosing one form of knowledge over the other for any reason would lead to philosophical falsehoods at worst, or a serious neglect at best. We believe that there has been such a neglect of procedural knowledge over the course of history, rooted in our human predilection for language on the one hand, and pictures on the other. Add to this that we are still puzzled over the nature of time. It has only recently, and reluctantly, been incorporated into our understanding of the physical world as an equal to space. Processes are hard to perceive, hard to understand, and, in the end, get the short end of the theoretical stick. We are much happier with observable measurement - the visual realm again - and static relationships between relatively fixed quantities.

This paper is a brief investigation of the nature of the duality between the two types of knowledge, and their representation for processing in computer form. Our preference for the conceptual graph formalism will lead us to a particular set of choices for representing procedural knowledge so that the perceived imbalance with declarative knowledge can be redressed.

2. A wish list

So that we do not lose sight of our goals, here is a set of requirements for representing procedural knowledge in a manner true to the spirit of conceptual graphs.

- use CG notation unaltered for declarative knowledge
- based on an ontology of action
- visual formalism for control flow (next action)
- ability to handle parallelism
- incorporating existing actor formalism for functions
- all standard programming language features: conditionals, looping, program state (memory), modularity.

Why these features are considered important will become clear during the course of the paper. We will not here try to present a completed design, only a specification. Hopefully this will come later after the specification has been analyzed and refined.

3. Types of knowledge and the history of their representation

We live in a visual world. Vision is the most refined of the human senses, so it is natural to use this sophistication to support some of our best efforts in theorizing about the world. Some of the biggest breakthroughs in the sciences have been made through visual metaphors. The snake eating its tail served as the model for the benzene ring. The unlocking of the mysteries of DNA were due in part to an ability to perceive the double helix as a possible solution. Even Einstein's thought experiments in relativity theory have a strong visual component. Space is far easier to perceive than time. Perhaps we should be more accurate and say that the extent of space is easier to perceive than the passage of time. It is certainly true that our children learn to handle the spatial world much sooner than they develop an ability to reason in time.

Writing is also visual, and therefore spatial in nature. Text has a linear structure as far as standard reading order goes (in Roman script languages, this is top-to bottom, left to right), but this is only because reading is a process that proceeds in time. Text itself can have any structure, as long as that structure can be mapped onto a three-dimensional space. However, if this structure is intended to capture non-linear time, then our cognitive limitations are revealed. Take a novel that attempts to use multiple flashbacks and out-of-time interludes. In general they are much harder to follow than a narrative that is strictly linear in time. Movies that take this approach are often less popular because they are harder to understand. Even the popular "Back to the Future" movies are presented in a simple linear time sequence even though the nominal times are different ("and then, after being in the past, we went back to 1985").

If we look at the kind of thought that results in this kind of linear presentation we find similarities. Thinking is at best linear in time, and is often supported visually. It is natural to do epistemology starting from the spatial/visual world and to add temporal stuff later. Knowledge representation has been like this, from the early days of Aristotle to the modern day. Whether we use logic, frames, semantic networks or rules, we are employing pictures and/or text (which is, after all, a form of pictorial representation) to capture things and their relationships. The absence of a temporal component is obvious. For instance, logic represents statements that can be true or false, but says nothing about the interval of time over which the relationship holds. Much of the work in artificial intelligence in this area has been aimed at addressing this issue. From non-monotonic logics that allow a statement to change its truth value (again they do not say when it changes) to full-fledged attempts to embrace time, research has been carried out to put time back into the picture [McCarthy, 80, McDermott, 80, Lin and Dean, 83]. All the other representational forms have made similar attempts to add a procedural component, but, it must be said, always after the initial structure was developed. The declarative side was always developed first, and the temporal component added later, sometimes as a after thought. This typically happens when a representation is first designed and applied to a particular domain. A declarative representation is just that: a set of declarations (of objects and their relationships). But what to do with it? How to reason with the relationships? And more importantly, how to represent and handle processes involving the objects?

The answer to these questions is the purpose of this paper. We seek not to merely add a procedural component to an existing declarative scheme, but to redress the historical imbalance between declarative and procedural representations.

4. A perceived imbalance

Procedural issues are commonly relegated to ways to change representations rather than to be representations of the actions themselves. Logic is stuck with logical inference, a set of syntactic rules that map onto commonly accepted natural inference results. Perhaps modus ponens is the most obvious and useful of these, but there are many others. Inference rules are permissive, which means that that can be applied whenever you choose. They are inadequate for capturing the intricacies of human thought, but have formed the basis of many problem-solving systems that do operate to produce a sequence of actions that mimic human problem solving in some aspects. Actually it is a testament to the true universality of the rules of inference that this is possible at all. We should marvel at a system that solves a puzzle just by proving a theorem, which is a way to characterize the applications of inference rules in order to achieve a desired result [Fikes and Nilsson, 71].

The generalization of inference into the rule set of an expert system also follows this pattern. Now a rule does not change the logical form of a representation, but a rule is allowed to change the state of objects and their relationships. The sequencing of the application of rules in the rule set can again be mapped onto sequences of actions, but again the temporal aspect arises out of the behavior of the rule engine, and is not represented explicitly. The attempt to represent and control the behavior of the rule engine (so-called control knowledge [Davis, 80]) did not really solve the temporal problem, since what was being represented was the rule engine itself (a program), not real-world actions.

Frame systems also fall into this category. Typically a frame system will do well in representing declarative relationships, some of an epistemological nature, such as the IS-A link, but then introduce actions, and therefore time, through add-on functions, or demons, which are programs much like the rules

in a rule system. In fact the hybrid knowledge representation system was very popular for a time, combining a declarative frame system and an attached rule system using the demon notion to provide procedural capability (KEE).

We perceive an imbalance between declarative and procedural issues, not only in the amount of effort applied, but also in the nature of the solutions. We believe this stems from the nature of human cognition, and its emphasis on visual, therefore spatial, and thus declarative issues.

5. Ontology

One approach to the whole issue of declarative versus procedural is to do some basic ontology. In an earlier paper [Hartley, 92] we made such an attempt, within the framework of conceptual graph theory. The results were quite encouraging then, although many details remained to be worked out. In summary, the world consists of objects and actions; spatial relationships between objects are declarative; temporal relationships between actions are procedural; case relations link objects to actions. This can be summarized in the "canonical square" (Figure 1).

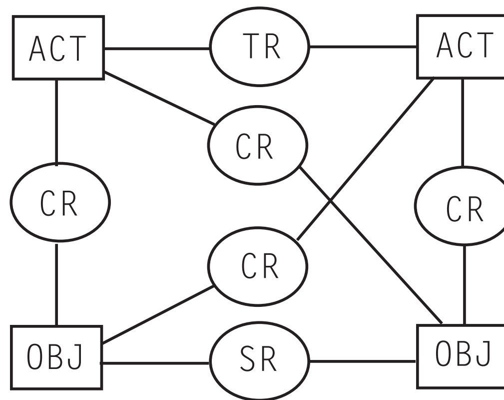


Figure 1. The canonical square.

This is nice because the duality of declarative and procedural knowledge is preserved in the symmetry of the ontology and the obvious visual symmetry in the CG notation. However, this is all still very declarative in nature. A temporal relationship appears as just another declarative relationship, begging the question of where the representation of procedural knowledge really is. The inadequacy is in the notation, not in the ontology. We hope to show how to fix this problem, while still preserving the visual nature of the CG formalism. We shall keep, however, the essential breakdown of objects, acts, and spatial and case relationships.

6. Visual representation is good

One way to look at conceptual graphs is that they are just another notation for logic, albeit a better one, mainly because of the ability to represent contexts in a perspicuous fashion [Mineau and Gerbé, 97]. This visual aspect of the CG notation has, we believe, been somewhat neglected [Hartley and Barnden, 98]. Moreover, if CGs are just a better way to use logic, then we have not made any progress towards the goal of handling procedural knowledge.

In fact, the only aspect of CG theory to address the issue head-on are the actors introduced in Sowa's first book [Sowa, 83]. Actors, however, only represent functions, but do not handle temporal relationships at all. Other attempts to handle procedural knowledge in a visual knowledge representation system suffer from the same problems mentioned in section 4. They are additions to the basic framework, not integrated into it. We could, however, point to a failed attempt by Rieger [Rieger, 76] as being truer in spirit to the set

of requirements listed in section 2. Rieger's analysis is an ontological analysis of actions and their conditions. Using a weird but wonderful variety of visual devices his formalism can represent the complex interplay of states and the actions that change them; enough at least to represent the complex physicla mechanisms, such as the reverse-trap flush toilet.

There are many other visual representations for procedures. Flow charts and petri nets are two such examples, the first for single threaded execution and the latter suitable for parallel processes. However, non of these methods address the ontological issues we think are vitally important, and are thus poor at declarative knowledge. Attempts to marry these techniques to conceptual graphs have, at best, been valiant failures, we believe [Lukose 97, Kremer 97].

7. Solutions

So, is there an answer? Is there a representation that meets the requirements listed in section 2 and might turn out to be a usable addition to the theory of conceptual structures? We believe there is, but let us first examine the candidates, and hopefully discover some additional constraints that we have not mentioned before. The following sub-sections examine these possibilities, in turn. They are:

1. Treat actions like objects
2. Using time to support temporal relationships
3. Programs

7.1. Treat actions like objects

This solution is essentially the one we mentioned before. Since declarative knowledge can be captured in static relationships, we can treat temporal relationships as similar. If a set of spatial relationships among objects could be called a snapshot, fixed in time, then a set of temporal relationships should be fixed in space. This fixed space is the union of all the spaces occupied by all the objects involved in the actions over the period of time during which the actions took place. In [Hartley, 92] we called the snapshot a schematic, and the "temporal snapshot" a chronicle. Whereas this is a neat idea, there was still no representation of the effect of actions *per se* past the addition of actors (the notion of a functional actor was extended to be almost synonymous with a rule) that changed states when they were fired. There was no obvious flow of time through the graphs, and, although representing parallel events and processes was easy, reading a large graph for an idea of "what happens next" was almost impossible.

7.2. Using time to support temporal relationships

In their visual form, CGs use real spatial relationships to support the uniqueness of objects. Two nodes in the same graph that contain the same concept label are assumed to represent different objects. Real spatial relationships are, however, represented by named relation nodes rather than using the relative placement of the object nodes to carry any meaning. Is it not possible for temporal relationships to be represented by a corresponding distinction in the time dimension, or at least to commandeer the third dimension to support the passage of time? The former idea might be supported by a movie consisting of a number of frames, each one carrying a snapshot of objects and their spatial relationships. Actions are then implicitly represented by the changes between frames. Unfortunately, this is rather like writing a program just by specifying the contents of variables only, leaving the assignment actions to be implicit. This is unwieldy at best. Perhaps better is to represent time with a third dimension (mapped, of course into a two-dimensional image). Work in visualization of program execution [Price et al, 93] may be useful here.

7.3. Programs

Computer programs are representations of a sequence of events (machine code instructions), where each event can be assumed to execute atomically. Clearly programs have the necessary dynamic characteristics to support procedural knowledge as long as long as we can use the ideas in programming languages to support the items in the wish list in section 2. Most programming languages are text-based, and the understanding of a program written in one of these languages relies on two factors mentioned before. Standard reading order says that events happen sequentially because one statement or command follows another down the page, and this order can only be subverted by special control-flow mechanisms which allow jumping from one part of the page to another. However, as we have already pointed out, using such a text-based language precludes the advantages offered by a visual representation, and in any case, there is no direct representation of control flow in these languages; control is represented declaratively, not procedurally.

Visual languages [Glinert, 90] add the visual counterpart by using spatial relationships (instead of merely reading order) and iconic devices, such as lines and arrows to support passage of time, and therefore to display flow of control. The dataflow paradigm used by many of these languages uses a functional model plus firing pre-conditions to do the same thing. Figure 2 shows a simple dataflow diagram of the arithmetic expression 3 * (4 + 5).

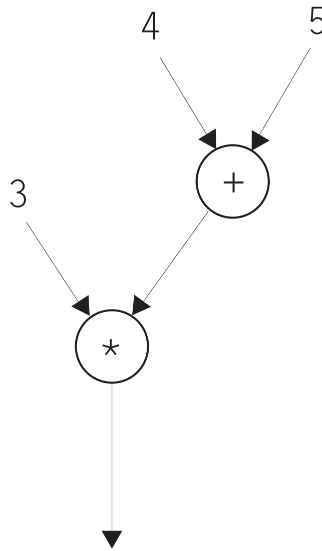


Figure 2. A simple dataflow expression.

In the example, each operation is considered to be firable only when all its inputs are present. The multiplication operation thus cannot fire until the addition operation has completed. Sequencing of operations is thus ensured because of the dependencies of operations on the outputs of other operations. The addition operation can fire because both its inputs are constants. In most of these dataflow models, inputs are consumed by the operation, which thus cannot fire gain until fresh inputs are supplied. In this pure dataflow model, there is no program state, since there are no variables to retain their values. Other models allow state variables to make programs more efficient even though they may be less pure. Text-based functional languages like Lisp and ML have related features to give variable-like facilities.

Another aspect of visual languages make them attractive as candidates for representation of procedural knowledge. This is the easy handling of parallelism, since an output of an operation may be split between two or more inputs to different inputs. Figure 3 shows such an example, where the output from the addition is split between two multiplication operations. This evaluates the two expressions 3 * (4 + 5) and 2 * (4 +

5) where the subexpression (4 + 5) is shared. Functional languages handle such a situation with a 'let' form which demands a name be introduced for the value of the sub-expression. The diagrammatic form has no

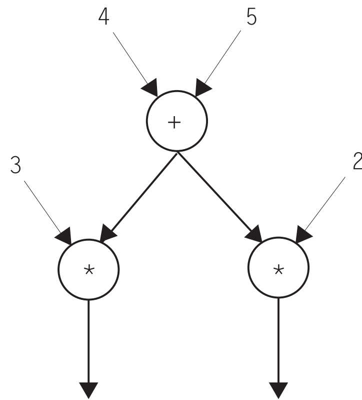


Figure 3. Parallel dataflow operations.

such problem, and the name is not needed.

These advantages, however, do not come without drawbacks. Although linear or parallel sequences of operations are well-represented, dataflow models have trouble with conditionals and loops. Of course, flowcharts, using a similar box and arrow notation, have been used to show control flow since the early days of computing. Showing conditionals and loops is no trouble for these conventions, since they are closely bound up with the idea, mentioned above, of jumping from one command to another according to the interpretation of some embedded test. Dataflow diagrams, in contrast, have an implicit notion of control embedded in the firing rules for operations. Two ideas help us out of the difficulties. One is the idea of passing Boolean values to a special 'gate' operation that passes its input to one of two outputs depending on the value of the Boolean input. Figure 4 shows how this can work for a test involving equality to zero.

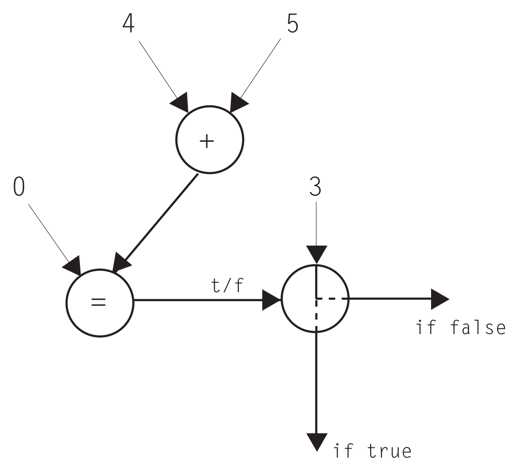


Figure 4. A dataflow conditional

.

The idea behind a loop can be captured using a generator. Since most loops can be classified as repeatedly applying an operation to a different operand each time round the loop (usually called the 'loop variable'), a generator can be diagrammed as shown in Figure 5.

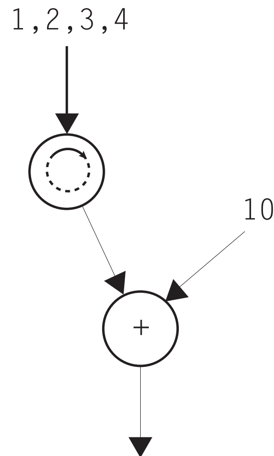


Figure 5. A dataflow generator.

Here the usual aspects of dataflow must be altered. The generator does not fire only once, since it must fire repeatedly until its set of input values is exhausted. This means that it must have state, i.e. a memory. Moreover its firings after the first are accomplished without an input (it effectively supplies its own). If these special properties are allowed for generator operations only, the rest of the dataflow model may be preserved.

7.4. Best guess

All of the preceding discussion comes to this: a way to merge the declarative CG model with the dataflow model to give visual representations that capture both kinds of knowledge in one notation, without the one affecting the other too greatly. Let us look first at a typical declarative representation of an action in CG notation. We will take as an example that is easy to understand, the task of making pancakes. Our aim will be to represent this in such a manner that all the relevant parts of the task are shown declaratively, but yet the representation will be executable as a program to represent the process in time from start to finish.

Figure 6 shows the action of putting flour into a bowl. A complete case analysis would involve agent, patient, source and destination, and perhaps others, but we will omit all agents in this example by assuming the same agent for all actions, and all irrelevant cases (such as the manner of the pouring, and the time at which it occurs). Our first change to the notation will be to put all actions (sub-types of the type ACT) in diamond boxes, just as standard CG actors are drawn. We will need to make a distinction between objects and actions right up front, since only actions can be executed as part of the conceptual program. These actions will be our dataflow nodes representing operations.

Next we will need to decide what data is, and how it flows through the graph. Clearly it is objects that flow, and we can make the case links do double duty and serve as dataflow links between the actions. Normally case relation links point away from their action, but we will reverse these to show the direction of data flow. Now (Figure 7) the object that is attached to the concept box flows along the link 'into' the action node. The concept label gives the type of object, and the case relation label is the 'parameter' name.

.

Of course, every action produces one or more outputs. Here we employ the RSLT relation as a generic output case relation. Here of course, the arrow points away from the action (Figure 8). The result of putting flour in a bowl is to make a 'mixture'.

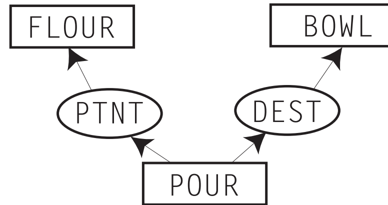


Figure 6. Pouring flour into a bowl

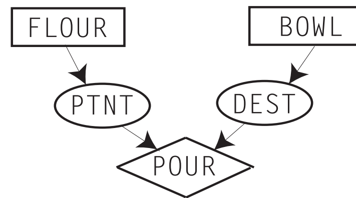


Figure 7.A dataflow actor with its inputs.

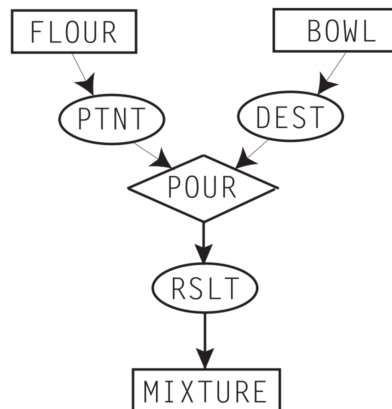
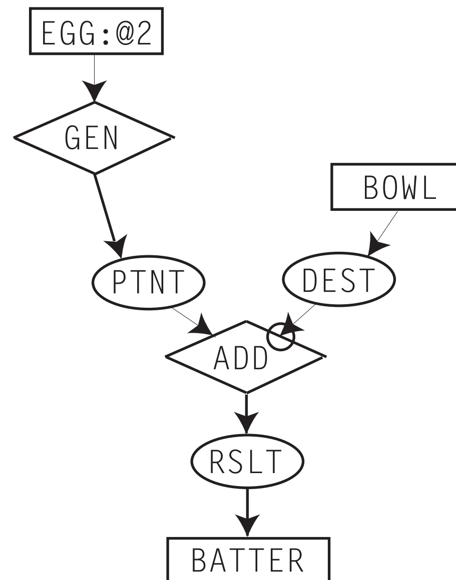


Figure 8. A completed actor with inputs and output.

Let us break two eggs into the mixture. Here we employ the generator node, starting with a set of two eggs, breaking each one and adding it to the mixture (Figure 9). A deficiency of the default firing scheme is revealed here. If we allow the ADD actor to fire with the first egg, there is no way it can fire again with the second egg since the mixture input will have been consumed. The generator can only remember how many eggs are involved. One way to handle this is to give any actor a 'memory' by optionally latching its inputs. If the mixture is latched when the actor fires for the first time, it will immediately fire again after the second egg is generated. The small circle on the input from the generator indicates a latched input.



Now we need to stir the batter for one minute, and if it is too thick, add a tablespoon of water. The use of the conditional 'gate' is shown in Figure 10.

The pancake batter is ready for cooking, but we will not show its representation due to lack of space, so we will turn instead to further discussion.

8. The missing link: methods

This we believe to be a reasonable proposal for handling procedural knowledge starting from declarative descriptions, such as CGs. However, the question is obvious: the graphs now say what is involved with an act like stirring the batter, but they do not say *how* it is to be done. They still show their declarative roots. The answer, we believe lies in a definitional mechanism for acts. Just as a declarative definition offers a more detailed view of a concept in terms of other objects and their spatial relationships with each other, and their case relationships with acts, a procedural definition should offer a detailed view of values and how they are 'computed' by an action. The standard functional actors can help out here. Figure 11 shows the 'method' definition of STIR (actually STIR-BATTER). This consists entirely of functional actors and dataflow operations. Essentially it is a procedure that might have been written in any programming language.

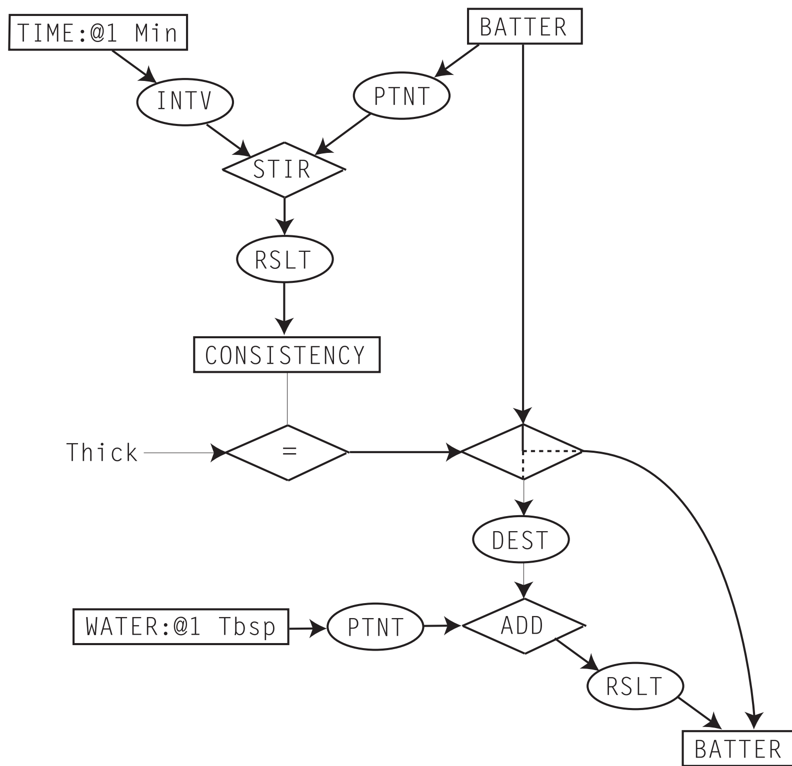


Figure 10. Stirring the batter.

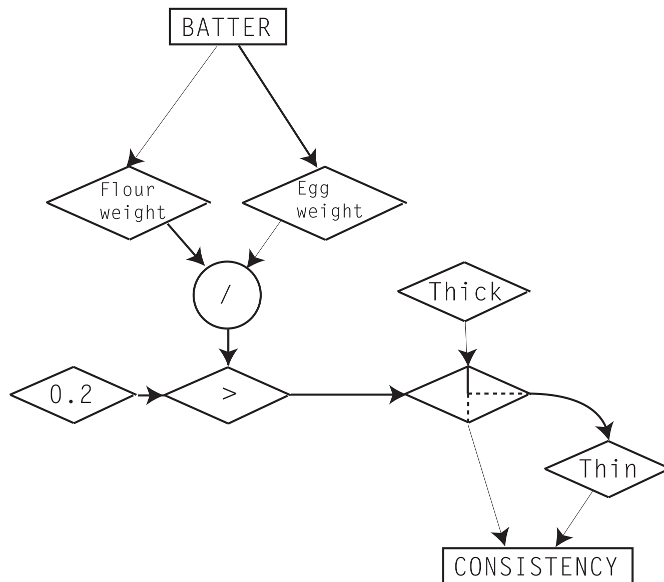


Figure 11. The definition of the STIR actor.

9. Conclusion

This paper presents a proposed solution to the questions stemming from the wish list in section 2. These are:

1. can we redress the imbalance in the representations of declarative and procedural knowledge?
2. can we do so without affecting standard CG theory too much?
3. can we emphasize the visual nature of CGs in the solution?
4. can we incorporate a usable ontology of action into the solution?

We believe all of these questions have been addressed, and a solution proposed which could be usable as a form of conceptual programming which can support many applications: simulation, reasoning, and knowledge representation. In particular, the addition of visual language constructs turn CGS into dataflow diagrams that are automatically convey the procedural nature of programs. Details not addressed here are the nature of time (moments *vs.* intervals, events *vs.* processes), and practical issues concerning the presentation and management of visual representations. These will be examined in further work.

1. References

Davis, Randy (1980) Meta-rules: Reasoning about Control. Artificial Intelligence Vol. 15(3).

Dean, Thomas (1983) Time Map Maintenance. Tech. Rep. 289. Yale University Computer Science Department.

Fikes, Richard, and Nilsson, Nils, (1971) STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, Artificial Intelligence Vol. 2, pp. 189-208.

Glinert, E. (1990) Visual Programming Environments (2 vols.) Los Alamitos, CA: IEE Computer Society Press.

Hartley Roger (1992) A Uniform Representation for Time and Space and Their Mutual Constraints. Computers Math. Applic. Vol . 23 No 6-9, pp. 441-457.

Hartley, Roger and Barnden, John (1997) Semantic Networks: Visualizations of Knowledge. Trends in Cognitive Science, Vol. 1(5) pp. 169-175.

Lukose, Dickson (1997) Complex Modelling Constructs inMODEL-ECS. Proc. Fifth International Conference on Conceptual Structures, Seattle. Lecture Notes in Artificial Intelligence 1257, Lukose, D., Delugach, H., Keeler, M., Sowa, J. (Eds.), Berlin: Springer.

Kremer, Robert, Lukose, Dickson, Gaines, Brian (1997) Knowledge Modeling Using Annotated Flow Charts. Proc. Fifth International Conference on Conceptual Structures, Seattle. Lecture Notes in Artificial Intelligence 1257, Lukose, D., Delugach, H., Keeler, M., Sowa, J. (Eds.), Berlin: Springer.

Lin, Shieu-Hong and Dean, Thomas (1994) Exploiting Locality in Temporal Reasoning, in Current Trends in AI Planning, E. Sandewall and C. Backstrom ed., Amsterdam: IOS Press.

McCarthy, John (1980) Circumscription-a Form of Non-Monotonic Reasoning, Artificial Intelligence, Vol. 13, pp. 27-39.

McDermott, Drew (1980) Non-Monotonic Logic I. Artificial Intelligence Vol. 13(1, 2) pp. 41-72.

Mineau, Guy and Gerbé, Olivier (1997) Contexts: A Formal Definition of Worlds of Assertions. Proc. Fifth International Conference on Conceptual Structures, Seattle. Lecture Notes in Artificial Intelligence 1257, Lukose, D., Delugach, H., Keeler, M., Sowa, J. (Eds.), Berlin: Springer.

Price, B. A., Baecker, R. M. and Small, I. S. (1993) A Principled Taxonomy of Software Visualization. Journal of Visual Languages and Computing, Vol. 4(3) pp. 211-266.

Rieger, Chuck (1976) An Organization of Knowledge for Problem Solving abd Language Comprehension. Artificial Intelliegnce Vol. 7pp. 89-127.

Ryle, Gilbert (1949) The Concept of Mind, Harmondsworth, UK: Penguin Books.

Sowa, John (1983) Conceptual Structures. Ney York: Addison Wesley.