C-Frame: Constrained Frame as A General Knowledge Representation Scheme

Harris Wang
School of Computing and Information Systems
Athabasca University, AB T6R 2E1, CANADA
harrisw@athabascau.ca

***Abstract* - When we celebrate the 50 years glorious history Artificial Intelligence (AI), we must celebrate each of its significant achievements. Among many great results that have been developed over the years, frame and formal system such as first order must be mentioned and further studied. Indeed, new scientific results must be built on existing ones. In this paper, we report a general knowledge representation scheme called constrained frame or c-frame for short, which is developed by applying logic to frame as constraints. We show that by putting frame and formal logic into a unified formal system, c-frame provides much greater power for knowledge representation and automated reasoning. We also discuss some of its potential applications in AI.**

Key words - knowledge representation; frame; universal logic; intelligent system modeling

I INTRODUCTION

Artificial Intelligence (AI) is concerned with understanding intelligent behaviors of humans, animals and machines, and concerned with designing as well as implementing artifacts that can present intelligent behaviors such as reasoning, learning, communicating, scheduling, planning, perception and acting intelligently in complex environments. Over the last 50 years scientists and technologists worldwide have worked diligently from different research directions in order to achieve these ultimate research goals. We thus have many well established areas of AI research with many interesting results.

Regardless the differences between those different approaches and results in different areas of AI research, there are two tasks common: formalization and manipulation of formal expressions obtained through formalization. Needless to say, logic-based approaches to AI are the most typical examples of formalization and manipulation, so are the well-known systems in AI history including LOGIC THEORIST [1], PROSPECTOR [2], MYCIN [3] and many others. In fact, everything people have been doing in AI and CS in general is somehow related to formalization: scientists devise formal languages and systems for formalizing procedures (imperative computer languages), facts and rules, as well as various ways of manipulating the formalizations.

Although over the last 50 years many concepts and theories have been developed to explain a variety of intelligent behaviors and building different kinds of intelligent artifacts, the importance of research in formalization remains unchanged.

In other word, we believe that it is still important for AI researchers to study formal systems for a better understanding and devise a new one, especially when we celebrating the 50 anniversary of AI. That is exactly what we are doing in this research, to evaluate some of greatest formalization schemes in AI history, and to unify them and devise a new one. The new scheme we report in this paper called c-frame, or constrained frame, which is developed by unifying frame, a classical knowledge representation scheme in AI, and logics. The logic we are using for c-frame, however, is neither the propositional logic, nor the first order one. It is a new logic called unified logic. As shall be seen, there are also some enhancements to the frame first developed by Minsky.

In the rest of this paper, we shall first give an overview of the logic, and then present our c-frame in detail. In conclusion, we will summarize the results and findings, including the advantages and potential applications of the new scheme.

II. OVERVIEW OF UNILOG

UniLog is developed from first order logic with many extensions. First, unlike some other logic systems, UniLog allows giving detailed descriptions for a world in form of constrained object hierarchies, and allows setting several parameters on a world for agents. These parameters are: (1) Time Scale (TS) which indicates at what kind of time scale the agent thinks and acts; (2) Belief Scale (BS) as a positive integer from which intervals for formalizing probabilities and necessities of logic truth are calculated; (3) Semantic Distance (SD(w1, w2)) as an agent action to calculate the semantic distance between two agent worlds; (4) Personal Preference to Similarities (PPS) which sets an agent-oriented standard for deciding whether two worlds are similar by calculating their semantic distance;
(5) Time Point (TP) as a clock in a world, based on which new agent beliefs, particularly those involving time, can be derived.

UniLog also allows to construct domains as multiple sets, and allows to attach a domain dx to an object variable x in three ways: x:dx, x-:dx and x+:dx. If a domain is attached to a variable in a predicate as x:dx, the predicate can only be true when object unified with x is in dx; In addition to x:dx, x-:dx requires the reasoning agent to remove the object from dx afterwards; Different from x:dx and x-:dx, x+:dx indicates the reasoning agent to add the unified object to dx, regardless whether the object is in dx or not.

In *UniLog*, we use sixteen quantifiers to capture much more quantity aspects of knowledge. These quantifiers are: (*) for universal, (!) for existential, (!n) for precise number existential, (!r) for precise ratio existential, (?n) for fuzzy

number existential, (?r) for fuzzy ratio existential, (!>=n) for number at least, (!<=n) for number at most, (!>n) for number more than, (!<n) for number less than, (!>=r) for ratio at least, (!<=r) for ratio at most, (!>r) for ratio more than, (!<r) for ratio less than, (?>) for many, and (?<) few of object.

Further, *UniLog* allows attaching truth labels to formulas. A truth label is a composition of world label ω , modal label (β), and temporal label (θ, τ), where, w refers to a defined a world model in from of c-frame, beta takes its value from [0, 1.0/BS .. 1] for probability of logic truth, value from [1 .. BS] for necessity of logic truth; theta is a temporal directive specifying semantics of "since", "until", "at time point of" and "during period of", τ is either a time point or time interval. In a UniLog formula $F^{\omega(\beta, \theta, \tau)}$, the world label ω indicates on which world a formula is asserted, the modal label β indicates either the probability or the necessity of the logic truth, while temporal label θ and τ together indicates in what time formula F is true.

In *UniLog*, inferences are generally considered as specific agent actions on UniLog formalizations, including setting and getting status and values of object references of a world model, checking status or properties of other agent actions, calling for other agents to carry out other designated actions, making changes to domains, and evaluating or satisfying UniLog formulas as well. There are three results from evaluating or satisfying UniLog formulas: successfully get value false (F), successfully get value true (T), or for some reason the evaluation or satisfaction is delayed (D).

Finally, to handle the dynamics of agent worlds, including actions, time and changes, we introduce three meta-predicate □, □ and ɪ. We assert an action *fx* is started by □*fx*, completed by □*fx* and in progress by ɪ*fx*. We introduce meta-action ↓, ↑ and ↕. We state some agent starts to make logic formula φ true with ↓φ, is making φ true with ↕φ, and has made φ true with ↑φ. We also introduce two pairs of special connectives ▶ and ◀, ▷and ◁ to formalize notions such as φ is true until, since, after or before *B* is true. With these extensions, inference rules for abduction, induction, analogy, generalization, specialization, classification, uncertain or fuzzy reasoning, and nonmonotonic reasoning, reasoning about actions, changes and time can be formally defined and applied to worlds or c-frames as models of worlds.

III. The Constrained Frame

Essentially, knowledge representation is to model a world, which can be either physical or mental. In that sense, knowledge representation has some close connections with system modeling in software engineering, such as that between Minsky's frame and object-oriented approach for system modeling. As such, when devising a new knowledge representation scheme, we naturally looked for answer from new development in system modeling. As a result, we adopted some new concepts and good ideas in object-oriented territory for our c-frame.

Indeed, over the past decade object-oriented as a methodology for computer system development has steadily gained its great popularity in software industries. As we have mentioned in the previous section, however, the present object models still cannot capture some important aspects of the world for computer agents, and hence need to be extended. In the following, we shall present some extensions to the ordinary object model and explore the reasons behind the extensions.

A. Extension one: attachments to variables

In a computer program, objects are often referred to by variables, whereas each variable denotes one or some memory cells in which data can be stored. Thus, it is necessary to answer the following questions about a variable: what data can it refer to? Does it refer to any data at a specific time? Does the data it refers to satisfy the agent? How can an agent get a data object for the variable? Although type systems can be used to partially answer the first question, no appropriate mechanisms in a simple object model can fully answer all these questions. In our c-frame we answer these questions by attaching domains, states and categories to ordinary variables.

Domains of variables - In our c-frame, domain is used as an addition to type to answer the first question. A domain can be considered as a set of data objects. It can be attached to a variable to indicate what data objects the variable can refer to. For example, in modeling a payment system, the variable representing employee's salary can/may refer to data only in a specific range.

Values outside the specific range may imply some hidden mistakes. Domains can be defined in many different ways. Given a domain *dx*, we can attach it to an object variable *x* by using statement *x: dx*. For example, the statement

i:[0 .. 500]

attaches domain [0..500] to variable i. In particular, given that *Dx* is a domain, we use *~Dx* to denote the set of all objects that are not in *Dx*. Basically, there are three operations an agent can carry out on the domain of a variable: to add objects into the domain, to remove particular objects from the domain, and to retrieve objects from the domain without removing them.

States of variables - To answer the second and third question, we allow object variables in our c-frame to have their states. At any time, an object variable has to be in one of the following five states:

- *Not assigned and not assignable because its domain is empty.* So there is nothing to be assigned to the variable unless readjusting the domain.
- *Assignable but not assigned yet.* At some stage, a responsible agent will be able to pick up an object from the domain to instantiate the object variable.
- *Assigned but no consistency checking.* It might be useless for other agents to use the assigned value because it may be changed after local consistency checking.
- *Locally consistent.* Other agents can go ahead to use the assigned value.
- *the current assignment is the only choice unless readjusting the domain.* If the current value is not the

right choice for other agents, some agent at a higher level of the object hierarchies has to backtrack.

If permitted, an agent can set or retrieve the state of a variable. For example, given an object variable x, calling *x.state*() in our c-frame means to retrieve the state of variable x, and calling *x.state*(*STATE*) means to set the state of variable x to *STATE*.

Categories of variables In a program, world model or frame in AI term, whether or not an agent can access an object variable, and how the agent can assign or reassign a value to a variable depend on what parts the object variable plays in the program or the world model. To capture this, in our c-frame we classify variables into the following categories:

- *Feature* object variables are those whose values decide the features of the world or problem. Feature object variables are similar to coefficients in mathematical programming. In the same world, feature object variables may vary in accordance with the task being carried out or the goal being achieved.
- *Passive* object variables are those to which an agent can never assign values by just choosing objects from their domains.
- *Constant* object variables are those such that once they has been assigned values, any further change will be prohibited.
- *Persistent* object variables are those whose values need to be kept for reuse. This property is necessary in modeling data or object base systems.
- *Perceptive* object variables are interfaces to agents outside the current world. If an object variable is categorized as perceptive one, some agents inside the world will watch for changes, and react to such changes.
- *Private, public* or *protected* object variables. These categories indicate access permissions to variables and the values they refer to.

An agent with an appropriate permission can check or change the categories of an object variable. For example, given object variable x, *x.categories*() could be used to check the categories of variable x, whereas *x.categories*(*CATS*) could be used to change the categories of x to *CATS* which can be one of the above categories. However, it is not recommended to change the access permissions of a variable in a dynamic way, thus *private, public* and *protected* are not included in a *CATS*.

In our c-frame we call variables attributed variables, to which domains may or may not be attached. To declare an attributed variable in our c-frame, one may use the following notion, for example

Passive private Int netIncome : [5400 .. 62000]

It specifies that variable *netIncome* refers to integers ranging from 5400 to 62000 inclusively; it is a private variable, and it is a passive variable so that the value it refers to can only be calculated from the values of other variables.

B.Extension two: add identity constraints

By using domain, one can state exactly what objects a variable may refer to in a world model or a program. For example, by defining variable *student.age* as

Int student.age : [16 .. 30]

One can say that the age of a student must be in between 16 and 30 inclusively.

However, one still cannot formalize the dependencies of variables. For example, the model in which variable *student.age* is declared might be for all sorts of students including that of primary schools, high schools, universities and vocational institutes. Therefore, it is necessary to have another variable to indicate the sort of a student, say *student.sort*, which can have value from domain [*primary, high - school, university, vocational*]. Thus, the actual range of values that variable *student.age* can refer to really depends on the value of variable *student.sort*. To capture such inter-dependent relationship between variables, we introduce the concept of *identity constraint* which describes some identity of a world model in a sense. In defining a c-frame or world model, we use the following syntax to declare its identity constraints:

identities := {< *identityConstraints* >}

where *identity Constraints* is a list of first order formulas, mathematical equations and/or inequalities. For example, given two variables x and y referring to sub-objects of a world, we can specify its identity constraint as:

identities := {$x > y$}

C. Extension three: introduce trigger constraints

To enable an agent to act autonomously, we introduce the concept of *trigger constraint* into the ordinary object model, which describes such a relationship between a set of logical formulas and a group of actions that once the logical formulas are true, and then the corresponding actions should be taken by some qualified agents. Trigger constraints in our c-frame have the same syntax as production rules, which have been introduced from AI into several object oriented programming languages. In our c-frame, trigger constraints can be specified as

triggers := {< *triggerConstraints* >}

where *triggerConstraints* is a list of trigger constraints which is in turn defined as:

(< *triggeringCondition* >) ➡ {< *actions* >}

in which *triggeringCondition* can be a list of first order formulas, mathematical equations and/or inequalities.

D. Extension four: let agents have goals

Acting in a world, an agent often has some goals to achieve. For example, a household robot may need to minimize the overall time needed to complete a list of household tasks. To capture this, we introduce the concept of *goal constraint* which can be specified in a world model as

goals := {< *goalConstraints* >}

where *goalConstraints* can be a list of first order formulas, mathematical equations and/or inequalities. In addition, a goal can also be in form of

minimize($f(x_1, x_2, \ldots, x_n)$)

or

$$maximize(f(x_1, x_2, \ldots, x_n))$$

where f is a mathematical function, and $x_1, x_2, \ldots, x_n$ are constant objects or variables referring to sub-objects in the world.

Having extended the ordinary frame in four ways as discussed above, we are now able to specify frames as models of some worlds in which some designated agents will act. In brief, a perfect object in constrained frame is featured with the following extensions:

- *Domains, states* and *categories* can be attached to variables, in order for agents to know what objects can be assigned to a variable, and whether it is allowed to change the value or state of a variable, and how to make such changes. It is often convenient to define a domain before using it.
- *Identity constraints* can be specified on variables referring to sub-objects, and used to check the suitability of assigned values of relevant variables. This is called *consistency checking*. The identity constraints can also be used to find a set of suitable values for the variables being constrained. This is called *satisfaction*. Because of the existence of identity constraints in a world model, once the values or states of some variables are changed, an agent can automatically adjust the values or states of other relevant variables.
- *Trigger constraints* can be specified and used to tell when or in what situation some actions should be taken. As such, the defined world model can be active and autonomous.
- *Goal constraints* can be specified and used to guide the corresponding agent to act towards some desired direction.

In addition, for the convenience of specifying constraints, it is necessary to introduce and use new relation symbols between composite objects. For example, we usually represent lines using two points, but we would like to simply state that "line A is longer than line B" instead of stating such relationship in terms of the values of two pairs of points. For this, we have to define a relation symbol *longer* before it can be used. Therefore, a perfect object as a model of some world can be formally defined as:

```
<c-frame>  ::=  <new domains to be used>
                <attributed variables>
                <actions or methods>
                <new relation symbols>
                <identity constraints>
                <trigger constraints>
                <goal constraints>
```

The following c-frame defines a common world for a Windows manager, who reacts to changes to some perceptive variables corresponding to keystroke or mouse motion, arranges windows without violating specified identity constraints, and to maximize free space of the root window for coming sub-windows.

```
World of Window {
  Bool activated; // indicate whether the window is focused.
  Point upLeft; //Given that world Point has been defined.
  Int width, height; // the width and height of the window.
  Passive Int area; // area = width * height.
  Perceptive Event event:[CLOSE, OPEN, RESIZE];.
  identities :={ area=width*height; }
} //end of Window

World of WindowManager {
  Constant Int width, height; // the geometry of root window.
  Window windows[]; // a list of sub-windows.
  identities:={
  windows[*].width<=width; //* means every.
  windows[*].height<=height;
  noOverlap(windows[...]); // no two windows overlapped.
}

triggers:={
  (windows[*].event=RESIZE)=>
  { windows[*].resize(); reSatisfy()};
  (windows[*].event=CLOSE)=>
  { windows[*].close(); reSatisfy()};
 (windows[*].event=OPEN)=>
  { windows[*].open(); reSatisfy()};
}

 goals:={ minimise(sum(windows[...].area)) };

} // end of WindowManager
```

Although we have left definitions of actions *resize*(), *close*(), and *open*(), goal action *minimize*(), and relation *noOverlap*() out in the above model, it still clearly and declaratively captures the main knowledge a window manager should have about its world.

IV. Conclusions

We have presented by now a generic knowledge representation scheme in which c-frames can be used to model worlds for computer agents. The scheme was developed from ontological investigations into the nature and generality of intelligent systems. We consider the works reported in [4–6] and [7–12] are relevant but with different focuses. We believe our scheme is more comprehensive and generic than others.

References

[1] Allen Newell, J C Shaw, and H A Simon: Empirical Explorations with the LogicTheory Machine: A Case Study in Heuristics, in Computers and Thought, 1963.

[2] Waterman A., Donald: A Guide to Expert Systems. Reading, Mass (USA).Addison-Wesley Publishing Company. pp 49-60, 1986

[3] Shortlifie, Edward H: CONSULTATION SYSTEMS FOR PHYSICIANS: The Role of Artificial Intelligence Techniques, In Readings in Artificial Intelligence (edited by Webber, Bonnie L. and Nilsson, Nils J), pp323-333. Tioga Publishing Company. Palo Alto, California, 1981

[4] Freeman-Benson B. N. and Borning A., Integrating Constraints with an Object-Oriented Language, Proceedings of the 1992 European Conference on Object- Oriented Programming, pp.268-286, 1992.
[5] Kanellakis P. C., Kuper M. G and Revesz P. Z., Constraint Query Languages. Journal of Computer and system science, 51:26-52, 1995.
[6] Ja.ar, J. and Lassez J.-L, Constraint Logic programming. Proceedings of the fourteenth ACM Conference on principles of programming languages. Munich. Association for Computing Machinery, 1987.
[7] Kinny D. and George., Modelling and Designing of Multi-Agent System, in J¨org P. M¨uller et al. (eds.) Intelligent Agent III, Springer-Verlag, LNAI 1193, pp.1-20, 1993.
[8] Myers B. M., Overview of the Amulet User Interface Toolkit, available from URL http://www.cs.cmu.edu/~amulet, 1999.
[9] Gus Lopez, The Design and Implementation of Kaleidoscope, A Constraint Imperative Programming Language, PhD dissertation, published as UW Tech Report 97-04-08 April 1997.
[10] Rist R., Terwilliger R., Object-Oriented Programming in Eifiel, Prentice Hall 1995.
[11] Claude Le Pape, Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-Based Scheduling Systems. in System Engineering, 1994, 3, (2), pp.55-66., 1994.
[12] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin and R. H. Badani, ECA Rule Integration into an OODBMS: Architecture and Implementation, Technical report N023, 1994, Department of Computer and Information Science and Engineering, University Florida, 1994.