Towards Situation-Oriented Programming Languages

Erkan Tın, Varol Akman
Department of Computer Engineering
Bilkent University
Bilkent, 06533 Ankara, Turkey
{tin, akman}@bilkent.edu.tr

Murat Ersan
Department of Computer Science
Brown University
Providence, RI 02912-1910, USA
me@cs.brown.edu

Abstract

Recently, there have been some attempts towards developing programming languages based on situation theory. These languages employ situation-theoretic constructs with varying degrees of divergence from the ontology of the theory. In this paper, we review three of these programming languages.

1 Introduction

The development of programming languages based on situation theory [1, 6] is a new trend. For this reason, it is worth examining how much these programming languages reflect situation-theoretic concepts and how much they deviate from them. In this paper, we review three approaches [9, 4, 12] towards programming systems based on situation theory, viz. PROSIT, ASTL, and BABY-SIT, respectively.

2 PROSIT

PROSIT (PROgramming in SItuation Theory), developed by Nakashima et al. [8, 9, 10], is *the* first situation-theoretic programming language. It is a declarative language in which both programs and data are just sets of declarative elements. This feature makes PROSIT akin to PROLOG, but PROSIT is based on situation theory [1, 2, 6] instead of Horn clauses. The motivation behind the design of this new language rests on the following features:

- The use of partially specified objects (e.g., situations) and partial information
- Situations as 'first-class citizens' of the theory
- Informational constraints
- Self-referential expressions

In PROSIT, an infon (a discrete item of information) is represented as a list whose first element is the relation and whose remaining elements are the arguments of the relation:

$(relation\ object_1\ \ldots\ object_n)$.

For example, the infon

```
(listening-to John Mary)
```

expresses that the relation *listening-to* holds between the objects represented by `John` and `Mary`, i.e., John is listening to Mary.

One can assert infons and make queries about them. Unlike PROLOG, all infons are local to situations. For example, to assert the infon mentioned above in situation `sit1` the following expression is used:

```
(!= sit1 (listening-to John Mary))
```

Expressions in PROSIT are LISP-style objects (i.e., atoms or lists). Atoms that are numbers or strings are considered to be *constants*. Symbols starting with a character other than "`*`" are *parameters*. They are used to represent things in the world, such as individuals, situations, and relations. Usually, different parameters correspond to different entities. Parameters can be used in any infon including queries and constraints; their scope is global. A third kind of expression is a *variable*. Variables are represented with symbols starting with "`*`". They can only occur in queries and constraints. They can stand for any PROSIT expression, yet their scope is local to the constraint or query they participate in.

In PROSIT, there exists a tree hierarchy among all situations, where the situation `top` is at the root of the tree. `top` is the global situation and the 'owner' of all the other situations generated. One can traverse the 'situation tree' using the predicates `in` and `out`. Although it is possible to make queries from any situation about any other situation, the result will depend on where the query is made. If a situation `sit2` is defined in the current situation, say `sit1`, then `sit1` is said to be the owner of `sit2`, or equivalently:

- `sit2` is a part of `sit1`
- `sit1` describes `sit2`

The owner relation states that if `(!= sit2 infon)` holds in `sit1`, then `infon` holds in `sit2`, and conversely, if `infon` holds in `sit2` then `(!= sit2 infon)` holds in `sit1`. So, `in` causes the interpreter to go to a specified situation which will be a part of the 'current situation' (the situation in which the predicate is called) and `out` causes the interpreter to go to the owner of the current situation.

Similar to the owner relation between situations there is the 'subchunk' relation. It is denoted as `([_ sit1 sit2)`, where `sit1` is a subchunk of `sit2`, and conversely, `sit2` is a superchunk of `sit1`. When a situation, say `sit1`, is asserted to be the subchunk of another situation, say `sit2`, it means that `sit1` is totally described by `sit2`. A superchunk is like an owner except that `out` will always cause the interpreter to go to the owner, not to a superchunk.

PROSIT has two more relations that can be defined between situations. These are the 'subtype' relation and the 'subsituation' relation. When the subtype relation (denoted by `(@< sit1 sit2)`) is asserted, it causes the current situation to describe that `sit2` supports `i` for every infon `i` valid in `sit1` and that `sit2` respects every constraint that is respected by `sit1`, i.e., `sit1` becomes a subtype of `sit2`. The subsituation relation is denoted as `(<-- sit1 sit2)` and is the same as `(@< sit1 sit2)` except that only infons, but no constraints, are inherited. Both relations are transitive. A distinguishing feature of PROSIT is that the language allows self-referential expressions. The fact that PROSIT permits situations as arguments to infons makes it possible to represent self-referential statements. Consider a card game where there are two players. John has the ace of spades and Mary has the queen of spades. When both players display their cards the following infons will be true:

```
(!= sit (has John ace-of-spades))
```

```
(!= sit (has Mary queen-of-spades))
(!= sit (sees John sit))
(!= sit (sees Mary sit))
```

As stated above, the notion of informational constraints is a distinguishing feature that encouraged the design of PROSIT. Constraints can be considered as a special type of information that generates new facts. They are just a special case of infons, and therefore are also situated. A constraint can be specified using either of the three relations =>, <=, and <=>. Constraints specified with => are forward-chaining. They are of the form (=> *fact* $head_1$ $head_2$ ... $head_n$). If *fact* is asserted to the situation then all of the head facts are also asserted to that situation. Constraints specified with <= are backward-chaining. They are of the form (<= *head* $fact_1$ $fact_2$ ... $fact_n$). If each of the facts from 1 to n are supported by the situation, then the head fact is also supported (though not asserted) by the same situation. Constraints specified with <=> should be considered as both backward- and forward-chaining.

Hence, if there is a constraint stating that everything that smiles is happy in situation `sit1`, viz.

```
(resp sit1 (=> (smiles *X) (happy *X)))
```

then asserting `(smiles John)` in `sit1` will force PROSIT to assert the following infon in `sit1` too:

```
(happy John).
```

2.1 PROSIT versus Situation Theory

The three major concepts of situation theory [1, 2, 6] are infons, situations, and constraints. Infons are the basic informational units and are denoted as $\ll$*relation*, $argument_1$, ..., $argument_n$, *polarity*$\gg$. Here *relation* is an n-place relation, $argument_1$, ..., $argument_n$ are objects appropriate for the respective places of *relation*, and *polarity* denotes the polarity (0 or 1). It is possible to use spatial and temporal locations in the argument places of relations.

PROSIT represents infons as lists and this is similar to the representation of infons in situation theory. PROSIT has no special polarity argument in infons, but uses the predicate `no`. Thus, (*infon*) represents a positive infon whereas (`no` *infon*) stands for the negation of that infon. The only deficiency of infons in PROSIT appears in the notion of spatial and temporal locations. In PROSIT, it is possible to use location indicating parameters in the argument places of relations, but this would be putting the individuals and locations in the same category. However, Devlin [6, p. 35] remarks that "...infons are built up out of entities called relations, individuals, locations, and polarities" and because the majority of real life 'facts' pertain only to a certain region of space and a certain interval of time, it is desirable to handle (spatial and temporal) locations separately.

As mentioned above, situations are first-class citizens of the theory. There is no clear definition of what a situation exactly is. Rather, a situation is considered to be a structured part of the Reality that the agent somehow manages to pick out (individuate). The only definition given at this level is that of the *supports* relation:

> s *supports* α ($s \models \alpha$) means that α is an item of information that is true of s.

However, it is desirable to have some tools to handle situations. Abstract situations are the mathematical constructs using which we can abstract analogs of real situations. They are more amenable to mathematical manipulation. An abstract situation is defined as a set of infons. Given a real situation, s, the set $\{\alpha \mid s \models \alpha\}$ is the corresponding abstract situation.

PROSIT has situational parameters that are used to abstract analogs of real situations. In that sense, they can be considered as abstract situations. They are associated with sets of infons. The definition of *supports* changes to:

> A situation s *supports* an infon if the infon is explicitly asserted to hold in the situation or can be proved to hold by application of forward-chaining constraints in the situation.

As a result, *supports* reduces to simple set-membership. So we can conclude that the situations in PROSIT are equivalent to abstract situations.

In situation theory, the flow of information is carried out by constraints. A situation s will carry information relative to the constraint $C = [S \Rightarrow S']$, if s: $S[f]$, where f anchors the parameters in S and S'. Hence, the information carried by s relative to C is that there is a situation s', possibly extending s, of type $S'[f]$.

PROSIT also supports the concept of constraints, but handles them in a different fashion. These come in three flavors in PROSIT: forward-chaining constraints, backward-chaining constraints, and forward- and backward-chaining constraints. (This classification cannot be found in situation theory.) Built up on this classification, the creators of PROSIT came up with new definitions:

> An infon is *supported* by a situation if the infon is explicitly asserted to hold in the situation, or can be proved to hold by application of forward-chaining constraints in the situation.
> An infon is *permitted* by a situation if the infon is deduced through application of backward-chaining constraints.

It seems that this classification has no philosophical basis, but is offered because of implementation requirements. In fact, both methods (forward or backward) result in the same answers to queries. However, forward-chaining incurs a high cost at assertion-time, and backward-chaining incurs a high cost at query-time. Additionally, forward-chaining requires more computer memory. So what the expression "an infon is permitted in a situation" really means is that, the infon is supported by the situation but there is either no need or no space to store it. On the other hand, if implementation strategies are considered, it is a good feature to have this kind of choice. It is left to the user to choose which kind of constraints to use. For example, forward-chaining constraints can be used in the design of applications where the results may not be predictable, and backward-chaining can be used in diagnostic problems.

There are two additional points on which the constraints of PROSIT have been criticized (cf. [11]). The first point is that PROSIT's constraints are situated infon constraints, i.e., the constraints are about local facts within a situation rather than about situation-types. Though this criticism seems to be valid, it is possible to simulate constraints that are not local to one situation (but are global). This can be achieved by introducing a situation which is global to all other situations and then asserting the constraint in this global situation. Because all other situations will be in this global situation, any constraint that is asserted here will apply to all situations. For example,

```
(!= (resp topsit
     (<= (!= *Sit1 (touching *X *Y))
         (!= *Sit1 (kissing *X *Y)))))
```

states that if, in situation **topsit**, there is a situation that supports a fact with the relation *kissing*, then that situation also supports a fact with the relation *touching* on the same arguments.

The second criticism is that it is not possible to model conventional constraints in PROSIT. However, none of the existing systems is capable of performing this either.

An important feature of situation theory is the existence of types. Types are higher-order uniformities which cut across uniformities like individuals, relations, situations, and spatial and temporal locations. Just as individuals, temporal locations, spatial locations, relations, and situations, types are also (higher-order) uniformities that are discriminated by agents. In this framework, relations may have their argument places filled either with individuals, situations, locations, and other relations or with types of individuals, situations, locations, and relations. For example, if an agent sees smoke he can conclude that there is fire, since he is aware of the constraint which links situations where there is smoke to those where there is fire. This constraint is not particular to a certain instance, but holds in general. Actually the constraint links types of situations, viz., smoky-type of situations to ones with fire.

The development of types necessitates devices for making reference to arbitrary objects of a given type. Therefore, for each type T, an infinite collection of *parameters* $T_1, T_2, \ldots$ is introduced. For example, IND_3 is an IND-parameter (parameters of type IND).

These parameters offer some computational power, but we need more than that. Rather than parameters ranging over all individuals, we need parameters that range over a more restricted class, e.g., all men kicking footballs. Such parameters are called *restricted parameters*. For example,

$$\dot{r1} = \dot{a} \upharpoonright \ll kicking, \dot{a}, \dot{b}, 1 \gg$$
$$\dot{a} = IND3 \upharpoonright \ll man, IND3, 1 \gg$$
$$\dot{b} = IND2 \upharpoonright \ll football, IND2, 1 \gg.$$

Once defined, $\dot{r1}$ ranges over all men kicking footballs.

In addition, it is possible to obtain new types using a parameter, s, and a set, I, of infons (in the form $[s \mid s \models I]$). For example,

$$[SIT \mid SIT \models \ll kicking, \dot{a}, \dot{b}, 1 \gg]$$

represents a situation-type where a man is kicking a football and

$$[\dot{a} \mid SIT \models \ll kicking, \dot{a}, \dot{b}, 1 \gg]$$

denotes the type of men kicking a football.

In PROSIT some of these are hard to achieve and some are not even possible. First of all, there is no typing in PROSIT. A variable can match any parameter or constant without due regard to types. In one of the previous examples we defined $\dot{r1}$ as a restricted parameter ranging over all men kicking footballs. Once defined, $\dot{r1}$ will represent this subclass of individuals. But in PROSIT it is not possible to make this kind of parameter definitions which can be used throughout a program. The only thing one can do is to pose queries on restricted parameters. All men kicking footballs can be queried using the following expression:

```
(AND (kicking *a *b) (man *a) (football *b)).
```

Although none of the variables above are restricted, the expression queries a restricted class of individuals.

PROSIT has no mechanism to define types either. As a consequence, one essential criticism on PROSIT is the lack of situation-types. We cannot define a situation-type explicitly, i.e., there is no corresponding expression for defining all men kicking footballs as in situation theory. On the other hand PROSIT can query a certain type of situation and put constraints between situation-types.

So the real problem is that it is not possible to restrict a parameter or to assign a variable to a certain type. This also makes it impossible to define argument roles. Nevertheless, this deficiency does not prevent us from making queries about restricted parameters or putting constraints between situation-types.

As described in the previous section, PROSIT has a tree structure among situations, but in situation theory there is no mention of a hierarchy. There is no subchunk relation either. However, this hierarchy of PROSIT turns out to be useful in modeling some problems regarding knowledge and belief.

One may ask why there are two different relations (owner and superchunk) doing very similar jobs. The major difference between these relations is not what the PROSIT manual [10] says, i.e., the predicate **out** will take the interpreter to the owner not to the superchunk. More importantly, the owner relation is defined between situations which are parent-child in the situation tree and the superchunk relation between two situations that are siblings in this tree.

The other two relations (subtype and subsituation) should also be examined carefully. At first glance, it seems that there is a similarity between these relations and the concept of inheritance in object-oriented programming. However, in PROSIT the supersituation inherits all the infons from the subsituation, whereas in object-oriented programming it is the subclass that inherits the properties and methods from the superclass. Accordingly, it can be concluded that either the direction of inheritance is completely different in two paradigms or that the terms subsituation and subclass should not evoke object-oriented concepts.

Next comes the question of where we can use these relations. The example given in the PROSIT manual uses these relations to classify the airplanes of type DC (DC-9, DC-10, and so on). But from the situation-theoretical point of view, it is not correct to consider airplanes of type DC as a situation. An agent does not individuate DC type of airplanes as a situation and DC-9s as a subsituation of that situation. These can only be considered as a class and its subclass. This example surely suits well to object-oriented programming, but not to situation theory. PROSIT should make a clear distinction between situations and classes.

One would be hard-pressed to find anything about inheritance, supersituations, and subsituations when one reads the essential documents on situation theory [1, 2, 6]. The only thing that seems related to these concepts is the *part-of* relation which is defined as follows:

> A situation s_1 is a part of a situation s_2 (denoted as $s_1 \trianglelefteq s_2$) just in case every basic state of affairs that is a fact of s_1 is also a fact of s_2.

However if $sit_1 \trianglelefteq sit_2$ is true, then the only comment we should make is that these two are defining the same situation, but sit_2 gives a more fine-grained description (using more infons) than sit_1.

A very important point to note is that there is no relation between the tree-hierarchy among situations and the super/subclass relation between situations. The super/subclass relation is defined between situations that are in the same level of the situation tree.

2.2 Solving Problems Using PROSIT

The main group of problems that PROSIT can handle is that of individual knowledge and belief in multi-agent systems, and common knowledge (mutual information). There are three main properties that enable PROSIT to simulate human-like reasoning. The first one is situated programming, i.e., infons and constraints are local to situations. The second is PROSIT's situation tree structure, using which one can represent nested knowledge/belief (e.g., "A thinks that B believes that C knows ..."). The third is the use of inconsistencies to generate new information. Two good examples for this type of problems are the "Three Wisemen Problem" [9] (which uses all of the three properties above) and the "Treatment of Identity" [8].

PROSIT proposes an important forward-chaining feature; viz., the assertion of uninstantiated variables. For example, when the constraint

```
(=> (grandfather *X *Y)
    (parent *X *Z) (parent *Z *Y) (male *X))
```

is fired with an infon, say `(grandfather John Mary)`, the system should assert the following infons:

```
(parent John *Z1)
(parent *Z1 Mary)
(male John)
```

where `*Z1` is an uninstantiated variable. However, while a very useful feature, this has not been implemented correctly. The current system only asserts the first and the third infons and also concludes that `(parent John Mary)`.

PROSIT is the first situation-theoretic programming language, and therefore is a valuable study. It provides most of the features of situation theory. Additionally, it offers some other tools such as the distinction between constraints (forward or backward) or the tree-hierarchy among situations.

When publications on PROSIT are compared, it will be noticed that the more recent papers deviate from the earlier papers which are more theoretic. Most features proposed in the earlier papers are not supported in the implementation, e.g., the implementation does not support *compound terms* and *labels*. In addition, the inheritance mechanism is totally different than the one proposed.

Self-referential expressions and situations as arguments of infons are two powerful features. These features can efficiently be used in representing knowledge and belief (e.g., "Three Wisemen Problem"). The owner relation and the superchunk relation are useful in modeling such epistemic problems. So, PROSIT is primarily aimed at the more general problems of knowledge representation and is closer to the world of logic programming than natural language processing.

3 ASTL

Black's ASTL (A Situation-Theoretic Language) is based on situation theory [4]. ASTL is aimed at natural language processing [4]. One can define, in ASTL, constraints and rules of inference over situations. An interpreter, a basic version of which is implemented in Common LISP, passes over ASTL definitions to answer queries about a set of constraints and basic situations.

ASTL allows of individuals, relations, situations, parameters, and variables. These definitions form the basic terms of the language. Complex terms are in the form of i-terms (to be defined shortly), situation types, and situations. Situations can contain facts which have those situations as arguments. Sentences in ASTL are constructed from terms in the language and can be constraints, grammar rules, or word entries.

The complex term *i-term* is simply an infon $\langle rel,\ arg_1,\ \ldots,\ arg_n,\ pol\rangle$ where rel is a relation of arity n, arg_i is a term, and pol is either 0 or 1. A *situation type* is given in the form $[param|cond_1 \ldots cond_n]$ where $cond_i$ has the form $param \models i\text{-}term$. If situation `S1` supports the fact that Bob is a young person, this can be defined as:

$\texttt{S1:} \quad \texttt{[S | S} \models \langle\texttt{young, bob, 1}\rangle\texttt{]}.$

The single colon indicates that `S1` supports the situation type on its right-hand side. The supports relation in ASTL is global rather than situated. Consequently, query answering is independent of the situation in which the query is issued.

Constraints are actually backward-chaining constraints. Each constraint is of the form $sit_0 : type_0$ `<=` $sit_1 : type_1, \ldots, sit_n : type_n$, where sit_i is a situation or a variable, and $type_i$ is a situation type. If each sit_i, $1 \leq i \leq n$, supports the corresponding situation type, $type_i$, then sit_0 supports $type_0$. For example, the constraint that every man is a human being can be written as follows:

$$\texttt{*S: [S | S} \models \langle\texttt{human, *X, 1}\rangle\texttt{] <=}$$
$$\texttt{*S: [S | S} \models \langle\texttt{man, *X, 1}\rangle\texttt{]}.$$

`*S`, `*X` are variables and `S` is a parameter. An interesting property of ASTL is that constraints are global. Thus, a new situation of the appropriate type need not have a constraint explicitly added to it. For example, assume that `S1`, supporting the fact that Bob is a man, is asserted:

$\texttt{S1:} \quad \texttt{[S | S} \models \langle\texttt{man, bob, 1}\rangle\texttt{]}.$

This together with the constraint above would give:

$\texttt{S1:} \quad \texttt{[S | S} \models \langle\texttt{human, bob, 1}\rangle\texttt{]}.$

Grammar rules are another form of constraints. An example grammar rule describing the utterance of a sentence consisting of a noun phrase and verb phrase can be defined as:

$$\texttt{*S: [S | S} \models \langle\texttt{cat, S, sentence, 1}\rangle\texttt{] ->}$$
$$\texttt{*NP: [S | S} \models \langle\texttt{cat, S, nounphrase, 1}\rangle\texttt{],}$$
$$\texttt{*VP: [S | S} \models \langle\texttt{cat, S, verbphrase, 1}\rangle\texttt{]}$$

where `cat` denotes the category of the construct, and `->` indicates that this is a grammar rule. This rule can be read: "When there is a situation `*NP` of the given type and situation `*VP` of the given type, there is also a situation `*S` of the given type."

3.1 ASTL as a Situation-Theoretic Language

At the heart of situation theory lie *schemes of individuation*, ways to classify the world into 'uniformities' discriminated by cognitive agents. Situations, relations, individuals, temporal locations, and spatial locations are the basic uniformities. The need for a mathematical representation of these uniformities resulted in what is known as *types*. Types correspond to the cognitive process of individuating or discriminating uniformities in the world. The ontology of situation theory has been extended further to include other uniformities such as infons, polarities, etc. In this respect, ASTL does not allow its objects to be of some type. Only situations can be declared to have a situation type. Other objects in the system are left untyped. This approach has particular consequences on the conception of relations and parameters which are explained in the sequel.

There are three characteristics of an infon in ASTL which should be evaluated from the standpoint of situation theory: *argument places*, *minimality conditions*, and *argument roles*.

For a reasonable treatment of infons both in conceptual and computational levels, each relation should have a limited number of argument places. Consider the relation *walking*. A reasonable assumption will be that this relation has four argument places: a walking agent, direction/destination, location of walking, and time of walking.

To have a formally well-defined infon, there must be a lower bound as to the number of argument places to be filled in an n-place relation. For example, there must be at least one argument place of the relation *walking* filled, namely the walking agent. Otherwise, the infon $\ll$*walking*$\gg$ would have zero information content. Minimality conditions are, then, necessary for a relation to provide an item of information. All argument places of a relation in ASTL are required to be filled.

Any object appearing as an argument of a relation must be appropriate for the argument role imposed by that argument place. Hence, *appropriateness* conditions must be defined for each possible argument place of a relation. This is generally done by forming a set of infons for an argument place which are supposed to be supported by the *world situation* for a given object. At the primary level, each argument role requires the appropriate object to be of some basic type. That is, each argument role is associated with a certain *type*, the type of the object that may legitimately fill that argument role. In a technical sense, appropriate conditions for an argument role are *complex* types having only 'the world situation' as their *grounding* situation. ASTL does not allow definition of appropriateness conditions for arguments of relations. The relation *walking*, for example, might require its walking agent role to be filled by an animate object. Such a restriction can be defined only by using constraints in ASTL. However, this requires writing the restriction each time a new constraint about *walking* is to be added. Thus, an ASTL relation can have any object in the system as one of its arguments. Having appropriateness conditions as a built-in feature would be better.

In situation theory, a fact and its dual should not both be supported by a situation. ASTL does not provide a mechanism, such as truth maintenance, to preserve coherence within situations. It is claimed that this is left to the user's control and that it can be achieved by specifying some special constraints in the ASTL descriptions. A constraint of the form

```
*S: [S | S |= ⟨actual, S, 0⟩] ->
        *S: [S | S |= ⟨*R, *A, 1⟩],
        *S: [S | S |= ⟨*R, *A, 0⟩]
```

is given by Black as an example for such constraints. However, this is not a solution to the problem of having *incoherent situations*. Moreover, this approach may be quite expensive for the user since maintaining coherence is a hard and complicated task and when left to the user, a large number of constraints must be written. What is worse is that consequences of allowing incoherent situations and reasoning over them may be drastic, e.g., it may lead to unintended models during computation. It seems that coherence, as a built-in notion, can hardly be embedded in an extension of the existing version of ASTL since it is not a syntactical matter and requires meta level control over the whole model.

Parameters are place holders in infons for the objects of some type and they are used to represent indeterminate objects in situation theory. In ASTL, there is no special treatment of parameters which are just atomic objects in the model. Parameters are only used in identifying situation types. Since there is no notion of types other than situation types in ASTL, a parameter can hold the place of any object. For this reason, any parametric object carries symbolic importance in this system.

In situation theory, parameters are used to achieve abstraction at the level of almost all object types, i.e., situations, individuals, temporal locations, etc. However, in ASTL, abstraction is only at the level of situations. There is no direct equivalent of properties in ASTL. Consider the abstraction for an object having the property of being happy in some situation s:

$$[X \mid s \models \ll happy,\, X,\, 1\gg].$$

In the current version of ASTL, Black tries to achieve this by allowing situation types with parametric infons. But this is not an appropriate way to use abstractions since one cannot abstract over other objects such as individuals, temporal locations, etc. (cf. *object type-abstraction* and *situation type-abstraction* in [6]).

As mentioned above, parameters are place holders for indeterminate objects in situation theory, hence yielding a form of abstraction over objects. The ties of these abstractions with the real world occur via a kind of assignment function called *anchor*. This function changes from one cognitive agent to the other, and from one perspective to the other of a single cognitive agent. Information content of an abstract object increases when its parameters are anchored to objects in the real world by an anchor. An anchor maps a parameter to a unique, appropriate object in the world. Technically speaking, a parameter must be anchored to an object of the same type since the parameter is a filler for an object having specific properties. The issues of anchoring to a unique object and anchoring to an object of the same type introduce technical difficulties in building a computational system. In case of ASTL, there are several points worth mentioning. Black proposes to consider anchors as situations (*anchoring situations*) having infons of the form $\ll$*anchor-to*, *label*, *term*$\gg$ and other related infons. Second, the current version of ASTL must be modified to use anchoring situations. This cannot be controlled by the user. The main reason is that whenever an anchoring occurs, the system must check whether the first argument of the relation *anchor-to* is a label and the second one is a term. Moreover, the system must assure that the parameter is anchored to only one object in that anchoring situation. Finally, type checking for both of the arguments is required. The crux of all these problems lies in ASTL's not having type-theoretic objects and not employing parameters as they are intended in situation theory.

Although one can define constraints between situations in ASTL, the notion of background conditions for constraints is not available. This means that *conditional constraints* are not available. However, this can be achieved by writing a set of conditions which must be satisfied for the constraint to qualify as an applicable one. These conditions will obviously be placed on the consequent part of each ASTL constraint since all ASTL constraints are used for backward-chaining.

Being in a larger situation gives one the ability of having information about its subsituations. The *part-of* relation of situation theory is used to build such a structure (i.e., *information nesting*) among abstract situations. ASTL does not have a mechanism to relate two situations so that one will directly support all the facts that the other does. While this might be achieved via constraints of ASTL, there is no built-in structure between situations.

Another issue is *circularity*; allowing situations as arguments of their own infons is common in situation theory. This is also possible in ASTL.

3.2 ASTL as a Programming Language

ASTL has been developed with the natural language processing and natural language semantics in mind. Still, it is possible to use it as a general knowledge representation language. It is in the class of declarative languages which are known to be suitable for knowledge representation. Advantages of employing declarative or procedural approaches in knowledge-based systems are still being debated. Both have been justified from perspectives of cognitive science and philosophy. Some researchers studied unified approaches. For the time being, declarative approach fits best for a situation-theoretic computational language, but one can also benefit from procedural knowledge. (PROSIT is a candidate for a unified framework since it is possible to use LISP statements as part of the language.)

Black shows that his system is sound, but he leaves its completeness formally unproved. Therefore, we are not sure if it is complete or not. The user should rely on the language and its computation mechanism. There are two aspects of the inference mechanism for which ASTL should be evaluated accordingly.

The first is the form of constraints. A typical user studying situation theory will not only want to investigate if an infon is supported by a situation, but also want to see if an infon is not supported by that situation. In other words, he would like to know if a situation is not of a certain type and then use this knowledge. This calls for negation in both query statements and constraints. An elegant way to do this is by having the appropriate syntax and semantics for the negation of *supports* relation, i.e., letting "$\not\models$" be used in these statements. Consider the following constraint:

```
*S: [S | S ⊨ ⟨paid-little, *W, *S, 1⟩],
*S: [S | S ⊭ ⟨has-other-income, *W, *S, 1⟩] ->
          *S: [S | S ⊨ ⟨poor, *W, 1⟩]
```

which expresses the form of reasoning "if I know that a worker is paid little and I do not know that he has other income, then I know that he is poor." Another example is:

```
*S: [S | S ⊨ ⟨exists, smoke, *S, 1⟩],
*S: [S | S ⊨ ⟨exists, air, *S, 0⟩] ->
          *S: [S | S ⊨ ⟨exists, fire, *S, 1⟩].
```

This says that "if I know that there is smoke in a situation and I do not know that there does not exist air in the same situation, then there is a fire in that situation." Note that negated forms of infons in propositions with $\not\models$ are the assumptions. They are accepted to be true by default, unless otherwise stated.

The other aspect is that of the chaining mechanism. ASTL constraints are all in the form of backward-chaining constraints. The user can only issue queries. However, an intelligent agent has the ability to not only acquire information about situations and obtain new information about them by being attuned to assorted constraints, but also act accordingly to alter its environment. Thus, having forward-chaining constraints as well would be better. In this way, new situations would be created, new infons would be inserted into situations, and consequences of new infons would be observed.

ASTL provides a simple user interface. The user writes ASTL definitions into a file which can be loaded in a Common LISP environment. Other than querying what situations support, the user has the opportunity to view some system features. ASTL is not an interactive language in the sense that a static definition is input to the system and the user can observe what can be inferred from these definitions. Moreover, one cannot assert propositions to the system: new propositions must first be added to the static description and then the system must be reloaded. This prevents the user from directly seeing the consequences of his propositions. An ideal system should be designed as if it were a cognitive agent instantly receiving information about its environment, making decisions upon, and acting accordingly. A system built with this view in mind would be dynamic since it would be responsive, and incremental since it would develop itself by learning more and more in time.

It is questionable whether the current version of ASTL can be updated vis-à-vis new (probably computational) constructs to be developed within situation theory. A few extensions to ASTL, especially in order to obtain abstractions, are proposed by Black. Even though these seem to be easily embedded in the current version of ASTL, they do not reflect bona fide semantic and syntactic constructs of situation theory; rather, they are synthetic substitutes, doability and semantic consequences of which are unknown.

4 BABY-SIT

BABY-SIT is a computational medium based on situations, a prototype of which is currently being developed in KEE (Knowledge Engineering Environment) [7] on a SPARCstation. The primary motivation underlying BABY-SIT is to facilitate the development and testing of programs in domains ranging from linguistics to artificial intelligence within a unified framework built upon situation-theoretic constructs [11, 12].

The computational model underlying the current version of BABY-SIT consists of nine primitive domains: *individuals*, *times*, *places*, *relations*, *polarities*, *parameters*, *infons*, *situations*, and *types*. Each domain carries its own internal structure:

- Individuals: Unique atomic entities in the model which correspond to real objects in the world.
- Times: Individuals of distinguished type, representing temporal locations.
- Places: Similar to times, places are individuals which represent spatial locations.
- Relations: Various relations hold or fail to hold between objects. A relation has argument roles which must be occupied by appropriate objects.
- Polarities: The 'truth values' 0 and 1.
- Infons: Discrete items of information of the form $\ll rel, arg_1, \ldots, arg_n, pol \gg$, where *rel* is a relation, arg_i, $1 \le i \le n$, is an object of the appropriate type for the ith argument role, and *pol* is the polarity.
- Parameters: 'Place holders' for objects in the model. They are used to refer to arbitrary objects of a given type.

- Situations: (Abstract) situations are set-theoretic constructs, e.g., a set of *parametric infons* (comprising relations, parameters, and polarities). A parametric infon is the basic computational unit. By defining a hierarchy between them, situations can be embedded via the special relation *part-of*. A situation can be either (spatially and/or temporally) *located* or *unlocated*. Time and place for a situation can be declared by *time-of* and *place-of* relations, respectively.
- Types: Higher-order uniformities for individuating or discriminating uniformities in the world.

This computational model is shared by the three modes of computation in BABY-SIT: *assertion mode*, *constraints*, and *query mode*.

4.1 Modes of Computation

4.1.1 Assertion Mode

This mode provides an interactive environment in which one can define objects and their types. There are nine basic types corresponding to nine primitive domains: ~IND (individuals), ~TIM (times), ~LOC (places), ~REL (relations), ~POL (polarities), ~INF (infons), ~PAR (parameters), ~SIT (situations), and ~TYP (types). For instance, if l is a place, then l is of type ~LOC, and the infon ≪*of-type*, l, ~LOC, 1≫ is a fact in the background situation. Note that the type of all types is ~TYP. For example, the infons ≪*of-type*, ~LOC, ~TYP, 1≫ and ≪*of-type*, ~TYP, ~TYP, 1≫ are facts in the background situation by default. The syntax of the assertion mode (cf. [11]) is the same as in [6].

Suppose `bob` is an individual, `sees` is a relation, and `sit1` is a situation. Then, these objects can be declared as:

```
I> bob:~IND
I> sees:~REL
I> sit1:~SIT
```

The definition of relations includes the *appropriateness conditions* for their argument roles. Appropriateness conditions define the domains to which arguments of a relation belong. Each argument can be declared to be from one or more of the primitive domains above. Consider the relation above. If we like it to have two arguments, the former being of type individual and the latter being of type situation, we can write:

```
I> <sees | ~IND, ~SIT> [1]
```

The number in square brackets indicates the minimum number of arguments that can be used with the relation. Hence, `≪sees, bob, 1≫`, for example, is considered to be a valid infon (i.e., *unsaturated infon*) in the system.

In order for the parameters to be anchored to objects of the appropriate type, parameters must be declared to be from only one of the primitive domains. It is also possible to put restrictions on a parameter in the environment. Suppose we want to have a parameter `E` denoting any individual that sees situation `sit1`. This can be done by asserting:

```
I> E = IND1 ^ ≪sees, IND1, sit1, 1≫
```

`IND1` is a default system parameter of type ~IND. `E` is considered as an object of type ~PAR such that if it is anchored to an object, say `obj1`, then `obj1` must be of type ~IND and the background situation must support the infon `≪sees, obj1, sit1, 1≫`.

Parametric types are also allowed in BABY-SIT. They are of the form $[P \mid s \models I]$ where P is a parameter, s is a situation (i.e., a *grounding* situation), and I is a set of infons. The type of all situations that Bob sees can be defined in BABY-SIT as follows:

```
I> ~SITALL = [SIT1 | w |= ≪sees, bob, SIT1, 1≫]
```

Hence, `~SITALL` is seen as an object of type ~TYP in BABY-SIT and can be used as a type specifier for declaration of new objects in the environment. An object of type `~SITALL`, say `obj2`, is an object of basic type ~SIT such that the background situation supports the infon `≪sees, bob, obj2, 1≫`.

Naming infons enables one to easily refer to them in expressions. For instance, the infon `≪sees, bob, sit1, 1≫` can be named `infon1` by the assertion:

```
I> infon1 = ≪sees, bob, sit1, 1≫
```

In BABY-SIT, a *situation browser* enables one to create situations, browse them graphically, add or delete infons, and establish hierarchies among situations. For example, the following sequence of assertions creates a situation `sit2` and then adds the infon `≪sees, bob, sit1, 0≫` into it:

```
I> sit2:~SIT
I> sit2 |= ≪sees, bob, sit1, 0≫
```

Variables in BABY-SIT are only used in constraints and query expressions, and have scope only within the constraint or the query expression they appear. A variable can match any object appropriate for the place or the argument role it appears in. For example, given the relation above, variables `?S` and `?X` in the proposition `?S|=≪sees, ?X, sit1, 1≫` can only match objects of type ~SIT and ~IND, respectively.

4.1.2 Constraints

Barwise and Perry identify three forms of constraints [1]. *Necessary constraints* are those by which one can define or name things, e.g., "Every dog is a mammal." *Nomic constraints* are patterns that are usually called natural laws, e.g., "Blocks drop if not supported." *Conventional constraints* are those arising out of explicit or implicit conventions that hold within a community of living beings, e.g., "The first day of the month is the pay day." They are neither nomic nor necessary, i.e., they can be violated. All types of constraints can be *conditional* and *unconditional*. Conditional constraints can be applied to situations that meet some condition while unconditional constraints can be applied to all situations.

A BABY-SIT constraint is of the form:

$$antecedent_1, \ldots, antecedent_n \ \{\texttt{<=}, \texttt{=>}, \texttt{<=>}\} \ consequent_1, \ldots, consequent_m.$$

Each $antecedent_i$, $1 \leq i \leq n$, and each $consequent_j$, $1 \leq j \leq m$, is of the form sit $\{\models, \not\models\}$ $\ll rel, arg_1, \ldots, arg_l, pol \gg$ such that rel and each arg_k, $1 \leq k \leq l$, can either be an object of appropriate type or a variable.

Each constraint has an identifier associated with it and must belong to a group of constraints. For example, the following is a backward-chaining constraint named HUMAN-BEINGS-012 under the constraint group SPECIES-PERSPECTIVE:

```
SPECIES-PERSPECTIVE:
  HUMAN-BEINGS-012:
    ?S |= ≪human, ?X, 1≫ <= ?S |= ≪man, ?X, 1≫
```

where ?S and ?X are variables. ?S can only be assigned an object of type ∼SIT while ?X can have values of some type appropriate for the argument roles of the human and man relations. This constraint can apply in any situation. Hence, BABY-SIT constraints can be global. Constraints can also be situated. For example, HUMAN-BEINGS-012 can be rewritten to apply only in situation sit1:

sit1 ⊨ ≪human, ?X, 1≫ <= sit1 ⊨ ≪man, ?X, 1≫.

Conditional constraints of BABY-SIT come with a set of *background conditions* which must be satisfied for the constraint to apply. For example, to state that blocks drop if not supported, one can write:

NATURAL-LAW-PERSPECTIVE:
  FALLING-BLOCK:
    ?S1 ⊨ ≪block, ?X, 1≫,
    ?S1 ⊨ ≪supported, ?X, 0≫ =>
        ?S2 ⊨ ≪drops, ?X, 1≫
  UNDER-CONDITIONS:
    w:≪exists, gravity, 1≫.

Background conditions are, in fact, assumptions which are required to hold for constraints to be eligible for activation. FALLING-BLOCK constraint can become a candidate for activation only if it is the case that $w \not\models$≪exists, gravity, 0≫, i.e., if the absence of gravity is not known in the background situation.

Forward-chaining mechanism of BABY-SIT is initiated either when the user tells the system to do so or by assertion of a new object into the system. A candidate forward-chaining constraint is activated whenever its antecedent part is satisfied. All the consequences are asserted if they do not yield a contradiction in the situation into which they are asserted. New assertions may in turn activate other candidate forward-chaining constraints. Candidate backward-chaining constraints are activated either when a query is entered explicitly or is issued by the forward-chaining mechanism.

In BABY-SIT, the following classes of constraints can be easily modeled [4]:

- Situation constraints: Constraints between situation types.
- Infon constraints: Constraints between infons (of a situation).
- Argument constraints: Constraints on argument roles (of an infon).

4.1.3 Query Mode

Query mode enables one to issue queries about situations. BABY-SIT's response depends on its understanding of the intention of the user. There are several possible actions which can be further controlled by the user:

- Searching for solutions by using a given group of constraints.
- Replacing each parameter in the query expression by the corresponding individual if there is a possible anchor, either partial or full, for that parameter provided by the given anchoring situation.
- Returning solutions. (Their number is determined by the user.)
- Displaying a solution with its parameters replaced by the individuals to which they are anchored by the given anchoring situation.
- For each solution, displaying infons anchoring any parameter in the solution to an individual in the given anchoring situation.
- Displaying a trace of anchoring of parameters in each solution.

The computation upon issuing a query is done either by direct querying through situations or by the application of backward-chaining constraints. A situation, s, *supports* an infon if the infon is either explicitly asserted to hold in s, or it is supported by a situation s' which is part of s, or it can be proven to hold by application of backward-chaining constraints. Given an anchoring situation, say anchor1, a query and the system's response to it are as follows:

Q> ?S ⊨ {≪sees, E, ?Y, 1≫,
        ≪time-of, sit1, ?Z, 1≫},
    w ⊭ ≪blind, bob, 1≫

answers (without anchoring of parameters):

sit3 ⊨ {≪sees, E, sit1, 1≫,
        ≪time-of, sit1, t1, 1≫},
    w ⊭ ≪blind, bob, 1≫

with the anchoring:

anchor1 ⊨ ≪anchor, E, bob, 1≫.

In addition to query operations, a special operation, *oracle*, is allowed in the query mode. An *oracle* is defined over an object and a set of infons (*set of issues*) [6]. The oracle of an object enables one to chronologically view the information about that object from a particular perspective provided by the given set of infons. One may consider oracles as 'histories' of specific objects. Given an object and a set of issues, BABY-SIT anchors all parameters in this set of issues and collects all infons supported by the situations in the system under a specific situation, thus forming a 'minimal' situation which supports all parameter-free infons in the set of issues.

4.2 Compatibility with Situation Theory

BABY-SIT accommodates the following basic features of situation theory:

- Objects: The world is viewed as a collection of objects. The basic objects include individuals, times, places, labels, situations, relations, and parameters.
- Situations: Situations are first-class citizens which represent limited portions of the world.
- Partiality: Infons can be made true or false, or may be left unmentioned by some situation.
- Coherence: A situation cannot support both an infon and its dual.
- Circularity: A situation can contain infons which have the former as arguments.
- Constraints: Information flow is made possible via coercions that link various types of objects.

Compared to the existing approaches [9, 10], BABY-SIT enhances the features listed above in the following ways:

- Situations are viewed at an abstract level. This means that situations are sets of parametric infons, but they may be non-well-founded (circularity) [3].
- Parameters are place holders and can be anchored to unique individuals in an anchoring situation. The anchoring situation is required to cohere.

- A situation can be realized if its parameters are anchored, either partially or fully, by an anchoring situation. That is, only anchoring the parameters of an infon contributes a piece of information about the situation.
- Each relation has 'appropriateness conditions' which determine the type of its arguments. The basic computation regime is unification.
- Situations (and hence infons they support) have spatiotemporal dimensions.
- A hierarchy of situations can be defined both statically and dynamically. A situation can have information about another which is a part of the former.
- Situations can be grouped to form a whole which provides a computational context. Such a whole has its own set of constraints which can be globally applied to the situations collected under it.
- Partial nature of situations facilitates computation with incomplete information.
- Constraints can be violated. This aspect is built directly into the computational mechanism: a constraint can be applied to a situation only if it does not introduce an incoherence.

BABY-SIT allows the use contextual information which plays a critical role in all forms of behavior and communication. Constraints enable one situation to provide information about another and serve as links between representations and the information they represent. Computation over situations occurs via constraints and is context-sensitive. In the existing approaches [4, 8, 9, 10], the notion of context is either poorly handled or left out completely. Furthermore, these approaches do not provide an apparatus for forming the background information which will assure the applicability of constraints. In BABY-SIT, the abstract nature of situations make it possible to form abstractions without asserting facts into them.

5 Conclusion

In various fields of science, one observes existence of well established theories that have been followed by their computational counterparts: fluid dynamics followed by computational fluid dynamics, geometry followed by computational geometry, and category theory followed by computational category theory. These computational fields of studies have been motivated by the foundations of the theories they are based on and they have led to useful systems which make basic and advanced features of their theories available to users. Situation theory is an obvious candidate in this direction [2, 5, 6].

The programming languages reviewed in this paper comprise initial attempts towards a computational account of situation theory. While they deviate from the ontology of the theory in varying degrees (cf. Appendix), they incorporate constructs tailored for efficient use in various domains of application ranging from artificial intelligence to natural language processing.

We also consider situation theory as a candidate framework for a new programming paradigm as justified by the nature of the existing approaches as general programming and knowledge representation languages. When we have a look at the history of programming language research, we find out that there are paradigms such as functional, logical, and object-oriented. Functional languages are motivated by λ-calculus (e.g., LISP), logical languages are based on first-order logic (e.g., PROLOG), and object-oriented languages are mainly built upon the concept of inheritance (e.g., Smalltalk). With its mathematical foundations based on intuitions basically coming from set theory and logic, situation theory adapts a remarkably original view of information, a logic, based not on truth but on information. We believe that this view of information together with situations as first-class objects are mature enough to establish a new programming paradigm whose computational flavor will be shaped by the existing and upcoming approaches.

Acknowledgments

KEE is a trademark of IntelliCorp, Inc. SPARCstation is a trademark of Sun Microsystems, Inc.

The second author's research is supported in part by a NATO SFS project (TU-LANGUAGE).

References

[1] J. Barwise and J. Perry. *Situations and Attitudes*, Cambridge, MA: MIT Press, 1983.

[2] J. Barwise. *The Situation in Logic*, CSLI Lecture Notes Number 17, Center for the Study of Language and Information, Stanford, CA, 1989.

[3] J. Barwise and J. Etchemendy. *The Liar: An Essay on Truth and Circularity*, New York, N.Y.: Oxford University Press, 1987.

[4] A. W. Black. "An Approach to Computational Situation Semantics," Ph.D. Thesis, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, U.K., April 1993.

[5] R. Cooper, K. Mukai, and J. Perry, editors. *Situation Theory and Its Applications*, Volume 1, CSLI Lecture Notes Number 22, Center for the Study of Language and Information, Stanford, CA, 1990.

[6] K. Devlin. *Logic and Information*, Cambridge, U.K.: Cambridge University Press, 1991.

[7] *KEE^{TM} (Knowledge Engineering Environment) Software Development System*, Version 4.1, IntelliCorp, Inc., Mountain View, CA, 1993.

[8] H. Nakashima, S. Peters, and H. Schütze. "Communication and Inference through Situations," in *Proceedings of the Third Conference on Artificial Intelligence Applications*, Washington, D.C.: IEEE Computer Society Press, 1987, pp. 76–81.

[9] H. Nakashima, H. Suzuki, P.-K. Halvorsen, and S. Peters. "Towards a Computational Interpretation of Situation Theory," in *Proceedings of the International Conference on Fifth Generation Computer Systems*, Institute for New Generation Computer Technology, Tokyo, Japan, 1988, pp. 489–498.

[10] H. Schütze. "The PROSIT Language v0.4," Manuscript, Center for the Study of Language and Information, Stanford University, Stanford, CA, 1991.

[11] E. Tın and V. Akman. "BABY-SIT: A Computational Medium Based on Situations," in P. Dekker and M. Stokhof, editors, *Proceedings of the 9th Amsterdam Colloquium*, Part III, University of Amsterdam, Amsterdam, Holland: Institute for Logic, Language, and Computation, 1993, 665–681.

[12] E. Tın and V. Akman. "BABY-SIT: Towards a Situation-Theoretic Computational Environment," in C. Martín-Vide, editor, *Current Issues in Mathematical Linguistics*, North-Holland Linguistic Series, Volume 56, Amsterdam, Holland: North-Holland, 1994, pp. 299–308.

[13] E. Tın and V. Akman. "Information-Oriented Computation with BABY-SIT," in *Conference on Information-Oriented Approaches to Logic, Language, and Computation (4th Conference on Situation Theory and its Applications)*, Saint Mary's College of California, Moraga, CA, 1994 (to be published by CSLI).

Appendix: Tableau comparison of existing approaches

Constraint Type	PROSIT	ASTL	BABY-SIT
Nomic	√	√	√
Necessary	√	√	√
Conventional	–	–	?
Conditional	–	–	√
Situated	√	–	–

Constraint Class	PROSIT	ASTL	BABY-SIT
Situation constraint	–	√	√
Infon constraint	√	√	√
Argument constraint	–	–	√

Computation	PROSIT	ASTL	BABY-SIT
Unification	√	√	√
Type-theoretic	–	–	√
Coherence	–	–	√
Forward-chaining	√	–	√
Backward-chaining	√	√	√
Bidirectional-chaining	√	–	√

Miscellaneous Features	PROSIT	ASTL	BABY-SIT
Circularity	√	√	√
Partiality	√	√	√
Parameters	?	?	√
Abstraction	?	?	√
Anchoring	?	?	√
Information nesting	√	√	√
Unsaturated infons	–	?	√
Set operations	√	–	–
Oracles	–	–	?

Legend
√: exists
– : doesn't exist
? : partially/conceptually exists