An automatic translator from KIF to PDDL

Fiona McNeill, Alan Bundy, Chris Walton

CISA, School of Informatics, The University of Edinburgh, Appleton Tower,
11 Crichton Street, Edinburgh, EH8 9LE, United Kingdom.
{f.j.mcneill,a.bundy,c.d.walton}@ed.ac.uk

October 4, 2004

Abstract

In this paper, we present a translation process that we have developed to convert KIF ontologies into PDDL. This allows us to define KIF-based agents that can plan efficiently. We discuss the difficulties inherent in such a translation process, and the steps we have taken to overcome them. This process is translates from only a subset of KIF to a corresponding subset of PDDL.

1 Introduction

It is a fairly common scenario that agents within a multi-agent communication system require the ability to form plans. However, in most circumstances, a different representation is required for the planner input than for the agent's internal ontology.

More generally, sharing and reuse of knowledge, and shared vocabularies are becoming increasingly important issues. If state-of-the-art PDDL planners can be made to be usable with knowledge that is not originally represented in PDDL, through translation processes, then they will be of much greater use to the broader AI community.

We are working with agents that have knowledge represented in KIF [3], which is a full first-order ontology language. However, KIF ontologies cannot be used as input for any state-of-the-art planner; full first order planners are inefficient and slow. Instead, we wish our agents to use a PDDL-based planner so as to able to plan efficiently. PDDL (Planning Domain Definition Language) is the language developed by the AIPS-98 Competition Committee for use in defining problem domains, and is a community standard for the representation and exchange of planning domain models [2]. We have therefore developed a translator that will convert the essential features of a KIF ontology into a PDDL representation. We can then use any PDDL based planner to produce a plan for achieving a given goal, and execute this plan within the KIF based agent system. This is currently only a one-way process: the translator will convert a KIF ontology into a PDDL representation so that planning can be performed, but the reverse translation from PDDL to KIF cannot currently be performed. We have not written a reverse translation back to KIF because this is not necessary in our system. The plan that is produced, the format of which will depend on the planner used, will need to be translated into a format that is readable by the KIF agent; however, this is a small problem because the format of the plan is not complex, but simply a sequence of actions.

KIF is a representation that supports full first-order logic, and PDDL is basically a first-order logic language [4], although PDDL only allows quantification over finite domains. It

is not possible to directly translate KIF statements that include quantification over infinite domains. However, the syntax of PDDL is developed so that it can be considered to be a first-order representation, with the added proviso that uninstantiated variables and quantification over infinite domains are not permitted. Hence, as long as we can deal with variables that are uninstantiated in the KIF ontology and exclude infinite domains, we can regard this process as a first-order to first-order translation process. By restricting the domain in such a way, a PDDL planner is able to unpack finite conjunctions and disjunctions in order to produce a propositional space to search through. This removes the overwhelming search problems faced by a true first-order planner.

The representation problems surrounding planning agents have already been recognised: an automatic translator between PDDL and DAML has been developed so that DAML agents can make use of PDDL planners [6]. DAML is not full first-order, so the translation problem is more constrained.

The aim of this paper is to describe the translator module of our ontological refinement system [7, 8]. We demonstrate that this translator, within the restrictions of our requirements, will produce PDDL files that correctly correspond to the original KIF ontology, and that can be processed by a PDDL planner to produce a plan that is executable within this KIF agent system. We are dealing with fairly simple KIF ontologies that do not utilise the full scope of the KIF language, and this translation process is only equipped to deal with such ontologies. This has the effect that, since the ontologies are relatively simple, the full capabilities of PDDL are not utilised either, because this is not necessary for full translations of our ontologies. It should be stressed that the motivation for this work is to provide a viable unit for our ontological refinement system; this problem is not being explored from a general point of view. Most significantly, the ontologies we are dealing with do not deal with temporal planning, and thus durative actions are not included. This paper discusses the issues surrounding the translation problem in order to describe how the problem has been tackled and what the difficulties are.

2 KIF and PDDL

There are six different types of ontological objects in a KIF ontology: functions, relations, axioms, classes, individuals and frames. Note that the term `function` has a slightly different meaning in KIF and PDDL. In KIF, a function refers to a kind of `relation` (or `predicate`), that, given instantiations for the first n-1 arguments, has a precisely determined value for the nth argument. On the other hand, a PDDL predicate for which the above holds is only referred to as a function if the nth argument is numerical [2]. A KIF `relation` corresponds to a PDDL `predicate`, with the exceptions stated above: PDDL `predicates` include those that are uniquely determined but non-numerical, whereas in KIF, these would be considered to be `functions` and not `relations`. For the sake of clarity, we refer to objects that are called `functions` in both KIF and PDDL (i.e. uniquely determined numerical functions) as `numerical functions`. A KIF `axiom` corresponds to a PDDL `action`; that is, a rule describing the preconditions and effects of a named action. KIF `classes` correspond to PDDL `types`. KIF `frames` and `individuals` both correspond to PDDL `objects`. A `frame` is an individual that has initial facts attached to it; an `individual` has none. The initial status of the problem is extracted from information contained within the frames and individuals of the KIF ontology.

Our KIF ontologies have been developed using the Ontolingua Ontology editor [1, 5]. This produces an HTML page containing the whole ontology, which can be saved to a single file. PDDL requires this file to be translated into a *domain* file and a *problem* file (see Figure 1 and the example below). In PDDL, the domain file contains information that is general to the whole domain: the names of predicates,

the numbers of arguments they take, the axioms, and so on. The problem file contains the information that is specific to a particular problem: the individuals, their classes, the facts and the goal. Hence a single domain file can be paired with several different problem files. In KIF the whole ontology is contained in a single file. Some types of KIF ontological objects are put in the problem file, and some in the domain file, because the KIF ontology contains not only a description of the domain, but also facts and individuals. Thus the current state; i.e., what is currently true, is represented by facts within the KIF ontology. After plan execution has been completed, the state will have changed. The KIF ontology is not kept up to date during plan execution; instead, the KIF ontology is updated once plan execution is complete, so that the state represented in the KIF ontology becomes compatible with the state achieved through execution of the plan.

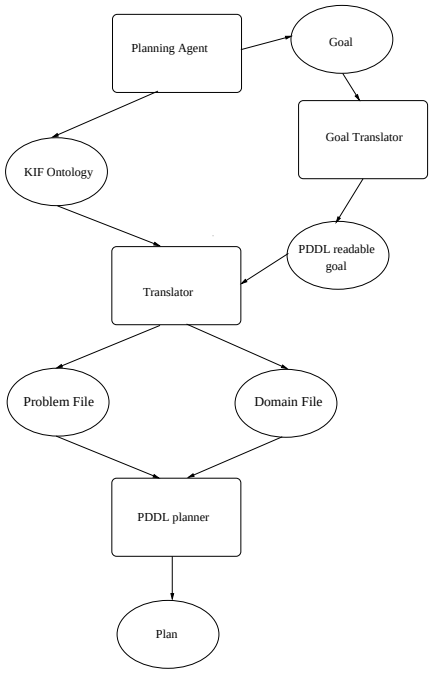

Figure 1: Architecture of Translation System

3 Motivating Example

Consider the situation in which a virtual travel agent is given a goal to purchase an on-line plane ticket. In order to achieve this goal, several steps must be carried out. For example, the agent must locate a ticket-selling agent, it must ensure it has sufficient funds, it must work out the correct origin and destination for the flight, and so on. Clearly, before the agent can act, it must have a plan for how to achieve the goal. Therefore, as soon as the agent identifies a goal, it sends the whole ontology, together with a suitable representation of this goal, to the translator. PDDL files for the ontology are produced, which can then be sent to the planner. The planner will produce a plan for how to achieve this goal, which can be translated into a format that is readable by the KIF agent. Once the KIF agent has the plan, it can then begin to execute the plan steps. In this short example, we have the following ontological objects in the original KIF ontology:

```
(Define-Frame Travel-Agent :Own-Slots
 ((Instance-Of Agent)) :Axioms ((Money
                  Travel-Agent 500)))

(Define-Frame Edinburgh :Own-Slots
 ((Instance-Of City)) :Axioms ((Flight
               Edinburgh London 300)))

(Define-Individual London (City) "")

(Define-Function Flight (?Place-0
 ?Place-1) :-> ?Value ""  :Def (And
     (Place ?Place-0) (Place ?Place-1)
                      (Number ?Value)))

(Define-Function Money (?Agent-0)
 :-> ?Value ""
 :Def (And (Agent ?Agent-0)
                   (Number ?Value)))

(Define-Class Agent (?X) ""
             :Def (And (Thing ?X)))

(Define-Class City (?X) ""
              :Def (And (Place ?X)))

(Define-Class Place (?X) ""
              :Def (And (Thing ?X)))

(Define-Axiom Book-Flight "" :=
 (=> (And (Flight ?Agent-Loc ?Conf-Loc
```

```
                          ?Price)
        (Money ?Agent ?Amount)
        (< ?Price ?Amount))
    (And (Has-Ticket ?Agent)
        (= ?Newamount (- ?Amount
                         ?Price))
        (Money ?Agent ?Newamount)
        (Not (Money ?Agent ?Amount))
     )))
```

There are objects referred to in the axiom that are not defined in the ontology section above: these are omitted for brevity.

Our translation would produce the following PDDL domain file from the above KIF ontology:

```
(define (domain domain Ont)
  (:requirements :strips :fluents :typing)

  (:predicates
    (Agent ?Agent)
    (Place ?Place)
    (City ?City)
  )

  (:functions
   (Money ?Agent)
   (Flight ?Place1 ?Place2)
  )

  (:action Book-Flight
    :parameters (?Agent ?City1 ?City2)
    :preconditions (And
                     (< (Flight
                        ?City1 ?City2)
                       (Money ?Agent))
                     (Agent ?Agent)
                     (City ?City1)
                     (City ?City2))
    :effects (And (Has-Ticket ?Agent)
                  (decrease
                    (Money ?Agent)
                    (Flight ?City1
                          ?City2)))
  ))
```

and the following PDDL problem file:

```
(define (problem problemOnt)
 (:domain domainOnt)
 (:objects London Edinburgh
              Travel-Agent
 )
 (:init
   (Agent Travel-Agent)
   (City London)
   (City Edinburgh)
   (= (Money Travel-Agent) 500)
   (= (Flight Edinburgh London) 300)
 )
 (:goal
   (Has-Ticket Travel-Agent)))
```

4 Translation Process

The translator is written in Prolog and works largely through pattern matching. For example, a key predicate is the `matchExpression` predicate, which takes a section of characters and an identifier that may or may not appear within that section and, if it finds the identifier, returns what comes before and after that identifier, and otherwise fails:

`matchExpression(-BeforeIdentifier, +Identifier,-AfterIdentifier,+Section)`

If the identifier appears more than once in the expression, the first appearance will always be used. In the above expression, following the Prolog convention, `+` indicates that this argument is instantiated when the predicate is called and `-` indicates that this predicate is uninstantiated when the predicate is called and is instantiated by the predicate. That is, `matchExpression` is passed an identifier and a section of code, and returns what comes before that identifier and what comes after. For example:

`matchExpression(Before,'Instance-of ', After,'(Instance-of Agent)').`

will return:

`Before = '('`

`After = 'Agent)'.`

4.1 Numerical Functions

The most significant difference between KIF and PDDL is the way that numerical functions are dealt with. The example ontologies in section 3 illustrate that the way in which KIF functions are defined does differentiate them from KIF predicates. The arguments of a

function are defined, e.g. `(?Place-0 ?Place-1) :-> ?Value` rather than simple as `(?Place-0 ?Place-1 ?Value)`. However, when a numerical function is referred to in a KIF ontology, either within an action or as an initial fact, it is dealt with not as a function but as a predicate. For example, a numerical function might be described as:
`(Define-Function Money (?Agent) :-> ?Amount ...,`
that is, as a function, but a possible instantiation would be:
`(Money Planning-Agent 100),`
so that it looks like a predicate.
In PDDL, the numerical argument is not included in the predicate definition, but rather it is written as a function, so that it would be stated:
`(Money ?Planning-Agent)`
and would appear within the `function` definitions in the domain file rather than in the `predicate` definitions.

The specific value of this function is not explicitly mentioned. The PDDL planner would be aware that this had a numerical value attached to it because it would be declared within *functions* rather than within *predicates*. Although it appears that information has been lost here, in fact the value of the function is tracked implicitly by PDDL; thus the information remains but it is no longer explicitly represented. If there is an instantiation for this numerical function in this initial state, then the value of this would be stated as follows:
`(= (Money ?Agent) 100).`

During the planning process, the PDDL planner will keep track of the value of all the predicates and these changing values are not referred to specifically within the axioms. However, in a KIF axiom, these values must be referred to and are thus given explicit names. For example, a `Buy` rule may have a precondition that the amount of money the buying-agent has must be greater than the cost of the item which is purchased. In KIF this would be stated as follows:
`(Money ?Agent ?Amount)` $\wedge$ `(Cost ?Item ?Price)` $\wedge$ `(> ?Amount ?Price)`
whereas in PDDL, this would be stated as:
`(> (Money ?Agent) (Cost ?Item))`

A postcondition for the same action might be that the money that the agent now has is the original amount less the cost of the item. In KIF, this would be:
`(= ?Newamount (- ?Amount ?Price))` $\wedge$ `(Money ?Agent ?Newamount)` $\wedge$ `(Not (Money (?Agent ?Amount))).`
In PDDL, this would be:
`(decrease (Money ?Agent) (Cost ?Item))`

Dealing with this difference in representation for numerical predicates is by far the most difficult aspect of the translation process. It causes some difficulties in writing the problem file, though these are not particularly hard to solve. More complex are the difficulties this creates in writing the domain file, and particularly in the statement of the axioms. These problems are discussed, together with our solutions to them, in sections 4.3 and 4.4.1.

4.2 Writing the Problem File

A PDDL problem file contains the specific details of this particular problem within the domain described in the domain file. The input for this process is the goal, and the list containing all the KIF definitions relevant for the problem file, which are those pertaining to individuals. The PDDL problem file needs to contain:

- A list of the names of the individuals
- A list of what is true initially, which includes:
  - A list of the classes of individuals
  - A list of the initial facts; i.e. initial instantiations of the predicates
- The goal

In KIF, facts are not stated independently but instead are attached to the first individual to which they pertain. For example, `(Location Agent1 Timbuktu)` would be contained either within the definition of the individual `Agent1` or within the definition of the information `Timbuktu`.

The information (contained in the list sent to the problem file) has not been processed at this stage, merely sifted for information relevant to the problem file. All the definitions within this relevant list are exactly as they appear in the KIF ontology. The first step is to process this list by extracting the useful information from the KIF definitions and forming it into three lists that correspond to the three items listed above (excluding the goal). This is done by searching for key markers within the definition. For example, an individual will either begin with the statement *Define-Individual*, if there are no facts attached to this individual definition, or *Define-Frame* if there are attached facts. The name of the individual always appears immediately after this initial marker. If the marker is Define-Individual, we need only extract the class of this individual. If the maker is Define-Frame, we then need to find the facts attached to this individual. These appear in two different places, depending on what kinds of facts they are. Some, including the class, which is indicated by the predicate *instance-of*, appear in a list soon after the name, and some appear in a separate list of axioms. The former do not include the name of the individual, which must be added in later.
Examples are given below:

```
(Define-Individual Isabelle-Paper-Dvi
      (Dvi-Paper))
```

This line, when processed, will add `Isabelle-Paper-Dvi` to the list of individuals and `(Dvi-Paper Isabelle-Paper-Dvi)` to the list of classes.

```
(Define-Frame Lucas :Own-Slots
  ((Has-Paper Isabelle-Paper-Dvi)
   (Instance-Of Agent)
   (Location Edinburgh))
  :Axioms ((Money Lucas 1000)))
```

This line will add `Lucas` to the list of individuals, `(Agent Lucas)` to the list of classes and `(Has-Paper Lucas Isabelle-Paper-Dvi)`, `(Location Lucas Edinburgh)` to the list of facts. The fact `(Money Lucas 1000)` will also be extracted from this line. However, because `Money` is a function, it requires further processing, and `(= (Money Lucas) 1000)` will be added to the list of facts.

Once the entire list of relevant definitions has been processed, the lists containing this information, together with the goal, are passed to a predicate which writes the problem file. This will first write the necessary initial information, such as the name of the problem that is being defined and the name of the domain within which the problem is described, to the problem file. The three lists (of individuals, classes of individuals and facts) are processed by simply writing them, member by member, within the correct brackets and initialisers. Finally, the goal, which has been translated from the Prolog format in which it was input to a format readable by PDDL, is inserted into the correct place.

4.3 Writing the Domain File

The domain file contains:

- Predicates, which includes:
  - all predicates that do not have a numerical value
  - class names
- Functions (non-numerical predicates)
- Actions, which contain the following information:
  - a list of all the variables mentioned in that action
  - the preconditions of the action
  - the effects of the action

The relevant lines of definitions are those defining KIF functions, relations, axioms and classes. As discussed above, KIF functions do

not correspond directly to PDDL functions, because PDDL only considers KIF numerical functions to be functions; non-numerical functions are considered to be predicates. KIF axioms correspond to PDDL actions. These input lines are processed to create four lists of information required by the domain file: a list of all the predicates (this includes both numerical and non-numerical predicates, i.e. both KIF relations and KIF functions, and both PDDL predicates and PDDL functions), a list of the classes, a list of the actions and a list of the numerical functions. The format of the list of all the predicates and the list of the numerical functions is different, because the former are represented as predicates and the latter as functions. In the latter, the predicates are listed with the numerical argument removed (which is how they must be expressed in PDDL, see section 2.2), whereas in the predicates list, because they are not identified as being numerical, they are listed as a predicate name, followed by a list of all the arguments and their classes. In the list of actions, each action is stored as an action name followed by a list containing all the preconditions, as they appear in the KIF ontology, and all the effects.

Writing the domain file is far more complex than writing the problem file, largely due to the difficulties with actions, which are discussed below. The file is initialised by stating the name of the domain file and the PDDL requirements. The predicate and class lists are adapted without too much difficulty so that they can be written down in the appropriate place. The numerical function list is used to write down the functions. Note that in our system at the moment, numerical functions are written down both as predicates and as functions. In the former version they have an extra argument (the numerical argument) which is not included when they are written as functions. It is fairly trivial to check predicates against the numerical predicate list and only write down those that are not numerical in the ordinary predicate slot. However, this is not done for reasons discussed in section 4.3.1. Expressing these numerical functions twice in different ways and in two different definition areas does not raise problems, as the planner considers them to be two different objects.

4.3.1 Pseudo Variables

One of the limitations of PDDL is that it cannot deal with uninstantiated variables. This is because, although PDDL appears to be a first-order language, most PDDL planners are in fact only pseudo-first-order, and work by creating all possible instantiations of the problem and searching through it, i.e. in a propositional manner. This is a problem for our system, as we wish to deal with agent plans in which there are unknowns after planning. For example, an agent may have a plan to attend a conference which involves registering at the conference and thereby receiving a registration number, and then using that number when actually attending the conference. Such confirmation numbers are useful in an agent system, as they allow the tracking of external objects that the agents possess, or privileges to which they are entitled. When forming a plan, it is not necessary, and indeed impossible, to know what these confirmation numbers are. These can only be instantiated during plan execution.

In order to force PDDL to deal with these uninstantiated variables, we have developed a class called `Confirmation-Number` and an individual belonging to that class called `Pseudo-Variable`. When writing an ontology, if we are creating an axiom in which a particular variable cannot be instantiated until plan execution, the individual `Pseudo-Variable` is inserted in place of this variable. This variable may or may not be numerical; that is, this pseudo-variable will sometimes be found in predicates that PDDL considers to be ordinary predicates, and sometimes in predicates that PDDL considers to be functions. However, if we are using `Pseudo-Variable` as a place holder in a predicate in some action, we do not want this predicate to be considered to be a function, since this means that PDDL will expect to be able to assign a specific numerical value to it.

When we are dealing with numerical func-

tions, we either want them to be considered as ordinary predicates, if the numerical argument is replaced by `Pseudo-Variable`, or as functions if it is not. The difficulty is that these `Pseudo-Variable` markers do not appear in the definition of the predicates, but only within specific actions. It is impossible to tell from the definition of a numerical function whether we will want to deal with it as a predicate or as a function. For this reason, since it does not create a problem with the planner, we define numerical functions as both predicates and functions (with one less argument), thus allowing PDDL to consider them as either, depending on the axiom it is currently dealing with.

4.3.2 Creating Actions

One of the more difficult tasks involved in writing the domain file is dealing with the numerical functions within the actions. In action definitions, it is not simply a case of inserting definitions. Instead, we must deal with arithmetic operations. An example of a KIF rule containing arithmetic operations, and its PDDL equivalent, are given below:
KIF rule:

```
(Define-Axiom Buy "" :=
  (=> (And (Price ?Item ?Cost)
           (Money ?Agent ?Amount)
           (Location ?Agent ?Shop)
           (< ?Cost ?Amount))
      (And (Has ?Agent ?Item)
           (= ?Newamount (- ?Amount ?Cost))
           (Money ?Agent ?Newamount)
           (Not (Money ?Agent ?Amount)))))
```

PDDL rule:

```
(:action Buy
  :parameters (?Item ?Agent ?Shop)
  :preconditions: (And (< (Price ?Item)
                          (Money ?Agent))
                       (Location ?Agent ?Shop)
                       (Agent ?Agent)
                       (Item ?Item)
                       (Shop ?Shop))
  :effects: (And (decrease (Money ?Agent)
                           (Price ?Item))
                 (Has ?Agent ?Item))
)
```

The first step is to alter the logic of the KIF to bring it in line with the logic of PDDL. That is, turn the KIF predicates into functions by removing the explicit representation of the value. For example:
`(Money ?Agent ?Amount)`
would be folded to the function:
`(Money ?Agent)`.

It appears that information has been lost in this process. However, the information contained in the variable `?Amount` still exists, it is just not explicit. PDDL tracks the values of all of the functions: a value will have been declared for `(Money ?Agent)` either initially or in a previous rule. The value contained in `?Amount` will be assigned implicitly to the PDDL function, and thus there is no need to represent it explicitly. However, we cannot immediately forget about the variable `?Amount`, because this will be used at other stages of the rule to refer to the value of `(Money ?Agent)`. It is still necessary to link these functions to the variable that represented their value, so that we know how these should be replaced within the arithmetic. `?Amount` is a marker for the value of `(Money ?Agent)`, and one can always refer to `?Amount` at any place in the KIF preconditions or effects of that action, and this will be a reference to the value of `(Money ?Agent)`. Thus, if we wish to change the amount of money, we can change the value of `?Amount` and assert this as the new argument of the predicate:
`(= ?NewAmount (- ?Amount ?Cost))` $\wedge$ `(Money ?Agent ?NewAmount)` $\wedge$ `(Not (Money ?Agent ?Amount))`.
When we treat these predicates as functions, we lose this value marker. In PDDL, it is not necessary to have a marker for the value of a function, because these values are automatically tracked by the planner. However, when we are translating to PDDL, we need to keep a record of these markers so as to be able to determine where these new functions should be placed. Thus, the following translation takes place:

$(f\ ?\vec{x}\ ?y) \wedge \Phi \Rightarrow \Phi\{?y/(f\ ?\vec{x})\}$

In the above expression, f indicates a function, $?\vec{x}$ indicates one or more variables, $?y$ indicates a single variable and Φ indicates the whole of the preconditions and effects. $\Phi\{?y/(f\ ?\vec{x})\}$ indicates the preconditions and effects, with every occurrence of $(f\ ?\vec{x})$ replaced by the variable $?y$; $?y$ is the marker for the function $(f\ ?\vec{x})$.

The first thing to be done is to strip all the predicates that will become numerical functions from the rule, keeping a record of their markers, and then replace any occurrence of these markers with the numerical functions. For example:

```
:preconditions (And (Money ?Agent ?Amount)
(Price ?Item ?Cost) (< ?Cost Amount)
(Location ?Agent ?Shop))

:effects (And (= ?NewAmount (- ?Amount
?Cost)) (Money ?Agent ?NewAmount) (Not
(Money ?Agent ?Amount)))
```

would first of all become:

```
Preconditions:  (And (< ?Cost ?Amount)
(Location ?Agent ?Shop))

Effects:  (And (= ?NewAmount (- ?Amount
?Cost))),
```

with stored information:
`[?Amount(Money ?Agent),?Cost(Price ?Item),?NewAmount(Money ?Agent)]`

The role in KIF of these predicates that have been removed is to create an identifier for the value. That is, by stating `(Money ?Agent ?Amount)` in the KIF preconditions, we have declared that `?Amount` is the temporary name given to the amount of money that `?Agent` has. In PDDL, such declarations are unnecessary because we do not need an explicit way of referring to the value. Thus these declarations are stripped from the preconditions. Note that we now have two different markers for the numerical function `(Money ?Agent)`, because the value of this function is changed by the rule. In KIF, there is no problem with having the same predicate with different markers, as the markers distinguish them. However, if we were to replace both these markers by the functions to which they are attached, we would have two occurrences of the same function, `(Money ?Agent)`, which would each time take a different value. For example, this would lead to statements such as:
`(= (Money ?Agent) (- (Money ?Agent) (Price ?Item)))`
which, since `(Price ?Item)` has a non-zero value, is not logically consistent. The reason these inconsistencies occur is because we have, at this stage, changed the logic but not changed the syntax. Since these predicates have now become functions, we have no need to assign values to them in the previous manner: we do not need an equality statement. For this reason, we do not replace markers that come immediately after an equals sign. Instead, we leave them in for this stage of the rewriting, and remove them later when the syntax is altered. So, after we have replaced the markers with the numerical functions, we have:

```
Preconditions:  (And (< (Price ?Item)
(Money ?Agent)) (Location ?Agent ?Shop))

Effects:  (And (= ?NewAmount (- (Money
?Agent) (Price ?Item))))
```

with stored information:
`[?NewAmount(Money ?Agent)]`

We now need to alter the syntax so that it is also in line with PDDL. There are three different types of operators that we need to consider: comparative operators, arithmetical operators and assignment operators. For comparative operators, the syntax of KIF matches the syntax of PDDL: once we have replaced the markers with the functions, we already have a readable PDDL comparator:
`(< (Price ?Item) (Money ?Agent))`

However, arithmetical and assignment operators are rather more complex. In KIF, assignment operators are always signalled by an equals sign, and the manner in which the assignment is being made is contained within the equality. For example,:

```
(= ?NewAmount (- ?Amount ?Cost))
```

means assign to the variable `?NewAmount` the value of `?Amount` less the value of `?Cost`. The arithmetical operator `-` gives further information about the way in which the value is assigned: in order to find the value of `?NewAmount`, we decrease `?Amount` by a certain amount. In PDDL, there are five assignment operators: `assign`, `scale-up`, `scale-down`, `increase` and `decrease`. So an expression in KIF that requires two arithmetical operators, `=` and `-`, can be represented in PDDL by a single operator, `decrease`. Likewise, an equality statement containing a `+` would correspond to `increase`, one containing a `*` would correspond to `scale-up`, and one containing a `/` would correspond to `scale-down`. We use these four assignment operators, as opposed to simply `assign`, because the function to which the value is being assigned is the same as one of the functions in the arithmetic expression: in this case, we are finding a new value for `(Money ?Agent)` by altering the old value by the amount represented by `(Cost ?Item)`. However, if we are assigning a value to a different function, we use `assign`. In our above example,

```
(= ?NewAmount (- ?Amount ?Cost))
```

will eventually become:

```
(decrease (Money ?Agent) (Price ?Item))
```

However, if the variable that was being assigned a value (in this case `?NewAmount`) did not correspond to a function within the equality statement, we would use `assign`. For example, if `?NewAmount` was a marker for a function `(Random-Value)`, the above statement would be converted to:

```
(assign (Random-Value) (- (Money ?Agent)
(Price ?Item)))
```

or perhaps `?NewAmount` refers to the money of another agent. We would then have:

```
(assign (Money ?Agent1) (- (Money ?Agent)
(Price ?Item)))
```

In this situation, because the arithmetical operator is not contained within the assignment operator, as it is in `decrease`, it must be used explicitly. Thus arithmetical operators are not always consumed by assignment operators; this depends on the situation.

Sometimes, KIF statements assign values to variables that do not correspond to functions at all. For example:

```
Preconditions:  (And (Price ?Item1 ?Cost1)
(Price ?Item2 ?Cost2) (Price ?Item3
?Cost3) (Money ?Agent ?Amount))

Effects:  (And (= ?Total (+ ?Cost1 ?Cost2
?Cost3)) (= ?NewAmount (- ?Amount ?Total))
(Money ?Agent ?NewAmount))
```

This is similar to the preconditions and effects of the rule above, except that we have a variable `?Total` which is a place holder for an expression, rather than a marker for a function. This is dealt with in a similar way to the function markers. The variable `?Total` is removed from the expression but information about what it is referring to is retained. It can then be inserted into the statement at a later stage. This would eventually lead us to:

```
Preconditions:  (And ())

Effects:  (And (decrease (Money ?Agent)
(+ (Price ?Item1) (Price ?Item2) (Price
?Item3))))
```

However, this would still not be correct PDDL. In KIF, the arithmetic function `+` can take two or more arguments, whereas in PDDL, `+` can only take exactly two arguments. Thus, if we find a `+` expression with more than two arguments, they must be nested. So the effects would become:

```
Effects:  (And (decrease (Money ?Agent)
(+ (Price ?Item1) ( + (Price ?Item2)
(Price ?Item3)))))
```

We have similar problems with the other arithmetical operators, and they are dealt with in a similar manner.

Although there are certain complications with the translation of preconditions and effects, some of which have been discussed above, it is nevertheless relatively straightforward to show that every case has been considered. There are a small number of KIF operators which correspond to a small number of PDDL operators and so, once the translation of some has been implemented, it is not difficult to generalise it so that it can apply to any KIF arithmetical statement.

Once the preconditions and the effects have been processed, all that remains to be done is to identify the variables used in the action, so that these can be declared. This is done simply by building a list of variables by stripping all the variables from the processed preconditions and postconditions, and then removing any duplicates from this list. This must be done after the preconditions and effects have been processed, as otherwise we will declare variables that do not appear in the processed preconditions and effects, such as `?NewAmount` or `?Total`.

Once these three lines of information: the variables (parameters), the preconditions and the effects, have been developed, the action can very easily be written down in the correct place in the file. All that remains is to locate the name of the action and place that in the proper place.

In summary, the main changes that need to be made are:

1. Remove the numerical argument from KIF numerical predicates, so that the predicate is folded into a PDDL function.

2. Remove all occurrences of that numerical predicate that do not appear in an arithmetical expression from the rule; these are there to assign values to the predicate, and are not necessary for PDDL functions.

3. Replace all occurrences of the marker (the name of the numerical variable in KIF) with the PDDL function.

4. Rearrange the arithmetic and the assignment operators accordingly.

5 Evaluation and Further Work

We have evaluated the translator from a purely practical point of view by plugging it in as a component of our dynamic ontology refinement system [7, 8]. In such a context, it is required to, and has proved capable of, automatically reading the KIF ontology, processing the ontology to produce the two PDDL files, passing these files to the planner, and receiving a plan from the planner. This plan is then interpreted and executed within the agent system. Frequent manual checks have been made to confirm that the PDDL files correspond correctly to the KIF ontology, and that the plan produced by the planner is indeed valid according to the KIF ontology. However, we do not claim that our translation process currently provides a full solution to the problem of translation from KIF to PDDL. The breadth of ontologies it has been tested on is not particularly wide. We know, for example, that some numerical operations have not been included in the translator because we do not currently have any need for them. More significantly, we are not dealing with temporal actions in our ontologies. The purpose of the current translator is as a working component of the system, rather than as an all-purpose KIF to PDDL translator. However, we believe that these missing operators could be added into the translator without much difficulty, and we anticipate that, with a limited amount of extra work, this translator could be made to translate from any finitely quantified KIF ontology to readable PDDL-1.2 files. We have not currently investigated how difficult it would be to translate ontologies that contain temporal actions and thus make use of the the extensions to PDDL contained in PDDL-2.1. We have also not investigated what could be

done with universal quantification in a KIF ontology so that some version of this ontology could be represented in PDDL.

The next stage of development for the translator would be to prove soundness and completeness for the translation process. As discussed previously, there is no question that the translation process is sound for the whole of KIF; there are many KIF expressions that cannot be represented under the current translation function, since we are only dealing with ontologies written in a subset of KIF. However, if we restrict the proof to a subset of KIF, then it should be possible to show that the rules of the translation process will take any KIF ontology within this subset and produce a logically valid PDDL representation. This can be proved by forming a Herbrand model of a KIF ontology and showing that this can be translated to a model of a PDDL representation that is logically equivalent to the KIF and also executable by a PDDL planner. This work will be undertaken in the near future and, since we are confident that the translation is sound for the subset of KIF with which we are working, this should not create difficulties. The next goal would be to widen the translation process to a larger subset of KIF, and eventually produce a translation function that is sound for the whole of KIF.

6 Conclusions

The aim of the work described in this paper is to create a component for our KIF-based ontology refinement system that enables our agents to use a PDDL planner. This aim has been achieved and the translator has been successfully tested on various ontologies. As described in the evaluation section, this development has been pragmatic rather than theoretical, and thus we do not claim that the translator is complete, but merely that it makes correct translations for the KIF ontologies we are working with. The problem of proving this translator to be sound and complete is discussed above.

References

[1] A. Farquhar, R. Fikes, and J. Rice. The ontolingua server: A tool for collaborative ontology construction, 1996. citeseer.nj.nec.com/farquhar96ontolingua.html.

[2] Maria Fox and David Long. An extension to PDDL for expressing temporal planning domains. Available from Durham Planning Group webpage:. http://www.dur.ac.uk/computer.science/research/stanstuff/planpage.html.

[3] M. R. Genesereth and R. E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Stanford, CA, USA, 1992.

[4] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl—the planning domain definition language, 1998.

[5] T. R. Gruber. Ontolingua: A mechanism to support portable ontologies, 1992. http://citeseer.ist.psu.edu/gruber92ontolingua.html.

[6] Drew V. McDermott, Dejing Dou, and Peishen Qi. An automatic translator between pddl and daml. http://www.cs.yale.edu/homes/dvm/daml/pddl_daml_translator1.html.

[7] F. McNeill, A. Bundy, and M. Schorlemmer. Dynamic ontology refinement. In *Proceedings of ICAPS'03 Workshop on Plan Execution*, Trento, Italy, June 2003.

[8] F. McNeill, A. Bundy, and C. Walton. Diagnosing and repairing ontological mismatches. In *Proceedings of the second starting AI Researchers' symposium*, Valencia, Spain, August 2004.