STAIRS 2012
K. Kersting and M. Toussaint (Eds.)
© 2012 The Authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License.
doi:10.3233/978-1-61499-096-3-282

OCL Plus: Processes and Events in Object-Centred Planning

Shahin SHAH [a,1] and Lukáš CHRPA [a] and Peter GREGORY [a] and Thomas. L. MCCLUSKEY [a] and Falilat JIMOH [a]

[a] *Department of Informatics, School of Computing and Engineering University of Huddersfield, UK*

Abstract. An important area in AI Planning is the expressiveness of planning domain specification languages such as PDDL, and their aptitude for modelling real applications. This paper presents *OCLplus*, an extension of a hierarchical object centred planning domain definition language, intended to support the representation of domains with continuous change. The main extension in *OCLplus* provides the capability of interconnection between the planners and the changes that are caused by other objects of the world. To this extent, the concept of event and process are introduced in the Hierarchical Task Network (HTN), object centred planning framework in which a process is responsible for either continuous or discrete changes, and an event is triggered if its precondition is met. We evaluate the use of *OCLplus* and compare it with a similar language, PDDL+.

Keywords. continuous planning, processes and events, object centered planning

1. Introduction

The control mechanisms of real-world planning problems need to be able to represent and reason with rich and detailed knowledge of such phenomena as movement and resource consumption in the context of uncertain and continuously changing environmental conditions [1]. Traditionally, physical systems with discrete and continuously-varying aspects have been represented using the mathematical notion of a hybrid dynamical system. This is a system that has a state made up of a set of real and discrete-valued variables that change over time according to some fixed set of constraints. Hybrid systems are used for modelling in applications such as embedded control systems [2].

The research-led standard domain model language in planning is PDDL (planning domain description language), which is based around a world view of parameterised actions and states, where it is assumed that a controller generates a collection of instantiated actions to solve some goal posed as state conditions. It has been extended to cope with real applications such as crisis management [4] and work-flow generation [13], and has versions which can represent time and resources [5]. More expressive modelling languages such as PDDL+ have been developed for applications where reasoning about

[1] Corresponding Author: Shahin Shah e-mail: s.shah@hud.ac.uk

processes and events in a mixed discrete/continuous world is necessary [6]. PDDL+ was recently used in an application for developing multiple battery usage policies [7]. Although PDDL is designed for logical precondition achievement, specialist forms of planning can be incorporated into the language using procedural attachment [3].

Despite its widespread acceptability, a serious problem with PDDL is that it reflects the concerns of those working in generative planning, rather than the execution and scheduling orientation of many applications. In contrast, scientists at NASA Ames developed the application-oriented language families HSTS [12] and then NDDL [11] for their applications in the Space arena. NDDL is fundamentally different to PDDL in that encodings are based around representations of objects and object instances, which persist in predefined timelines of continuous activities. Each activity has a start and end time interval (to represent uncertainty of duration), and the distinction between *action* and *state* is effectively blurred. Plan generation and execution are therefore linked to a much greater degree than with PDDL. NDDL has features to represent uncertain lengths of activities, though it does not support the representation of continuous processes. NDDL's concept of timelines are related to the idea of crafting abstract plans as in the input languages to HTN (Hierarchical Task Network) systems [8]. The idea of pre-written hierarchical plans to formulate possible behaviours has long been a popular type of formalism in which to encode dynamic knowledge for AI applications.

This paper describes PhD research motivated by knowledge formulation for automated planning and scheduling. Although the concept of automated planning in a continuously changing environment has been here in the AI planning community for decades, and has been taken seriously by many researchers, the central problems of designing effective representation languages and planning engines in this area still remain. We have adopted the concept of event and process, and found that these two powerful components can form the basis for modelling domains with continuous changes. In this context, we have extended the existing OCL_h to encode such domain models, calling it *OCLplus*, originally proposed by McCluskey [10]. *OCLplus* is derived from GIPOs [8] Object Centred Language (*OCL*) [9]. OCL_h is a structured formal language to acquire HTN type domain models. The main thrust of OCL_h is to identify the potential states of any object before the operators are defined. *OCLplus* is more expressive than its previous version as it supports continuous behaviour in the object centred HTN domain modelling. The main extension in *OCLplus* is the temporal attribute of a process which may or may not be interrupted by an event. In *OCLplus* time is modelled explicitly as a real quantity, like in PDDL+.

In the first part of the paper we start by re-visiting the constructs of OCL_h. Next we define the planning for continuous changes. In the third section of the paper we define the *OCLplus* for event and process. Finally, we discuss our progress, compare the language with PDDL+, and describe our plans for future research.

2. Background and Terminology

In this section we provide an overview of the background to this work. We will discuss existing formalisms for representing continuous planning problems and approaches used to solve planning problems for continuous changes. We first, however, provide a background of the *OCL* formalism (Complete description of *OCL* formalism can be found in [9]), on which the *OCLplus* language is an extension.

2.1. The OCL Planning Formalism

In the OCL_h a specification of the model $\mathscr{M}$ of a domain of interest $\mathscr{D}$, which consists of sets of:

- object Identifiers: *Objs*
- sort defintion: *Sorts*
- predicate definition: *Prds*
- substate class expression: *Exps*
- Invariants: $Invs^+$ (Positive Invariants) and $Invs^-$ (Negative Invariants)
- Operators: *Ops*

Definition 1 (Object Identifier) *An object identifier is a unique term that refers to a particular object in* $\mathscr{D}$.

Objects (*Objs*) in a model are classed as dynamic or static as appropriate - dynamic objects are those that may have a changing truth value throughout the course of plan execution, and dynamic objects are each associated with a changeable state.

A *ground, dynamic object description* is specified as a tuple (s,i,e), where i is the object's identifier, s is the primitive sort of i, and e is its substate, a set of ground dynamic predicates that all $refer$ to i.

Definition 2 (Object Expressions) *An object expression (oe) is a generalisation of a object description, and is specified using dynamic and possibly static predicates.*

In OCL_h is the idea of an **object expression** is crucial. Goals and operator preconditions are written as collections of object expressions. To define object expressions (***oe***) we need to introduce some notation that will be used throughout the paper.

- A legal substitution is a sequence of replacements, where each replacement substitutes a variable of sort s by a term which has s as either its primitive sort or its supersort.
- A set of static predicates are *consistent* if there is a legal substitution that instantiates them to facts asserted as true in the OCL_h domain model.
- If $p \subseteq \mathscr{P}$ then let $dyn(p)$ and $stc(p)$ be the dynamic and static predicates in p, respectively.

If $oe \subseteq \mathscr{P}$, then (s,i,oe) is called an **object expression** if there is an $ss \in substates(j)$ for some object identifier j of primitive sort s', and a legal substitution t such that

- $i_t = j$
- $dyn(oe)_t \subseteq ss$
- $s' = s$ or s' is a subsort of s
- $stc(oe)_t$ is consistent

In this case object (s',j,ss) is said to *satisfy* (s,i,oe). Since i could be a dynamic object identifier or variable, we refer to it as an *object term*.

Definition 3 (Sort) *A sort is a set of object identifiers in* $\mathscr{M}$ *denoting objects in* $\mathscr{D}$ *that share a common set of characteristics and behaviours. Sorts are either primitive or non-*

primitive. Non-primitive sorts are defined as the union of objects from two or more other sorts. A sort is primitive if it is not defined in terms of other sorts.

Sorts in OCL_h can be hierarchical. A sort hierarchy with object identification is shown in Example 1, containing 4 dynamic primitive sorts *truck, package, train, traincar*.

```
    sorts(physical_obj, [vehicle, package])
sorts(vehicle, [railv,roadv])
sorts(roadv, [truck])
sorts(railv, [train, traincar])
sorts(location, [city_location,city])
sorts(city_location, [tcentre,not_tcentre])
sorts(tcentre, [train_station])
sorts(not_tcentre, [clocation,post_office])
sorts(route, [road_route, rail_route])
objects(train_station, [city1_ts1,city2_ts1,city3_ts1])
objects(clocation, [city1_cl1,city1_cl2,city2_cl1,city3_cl1])
objects(post_office, [post_1])
objects(city, [city_1, city_2, city_3])
objects(train,[train_1])
objects(traincar,[traincar_1])
objects(road_route, [road_route_1,road_route_2,road_route_3])
objects(rail_route,[rail_route_2,rail_route_3,rail_route_4 ])
objects(truck, [truck_1, truck_2, truck_3])
objects(package,[pk_1, pk_2])
```

Example 1: A simple sort hierarchy

Definition 4 (Substate Class Expression) *A substate is defined as a set of ground, dynamic predicates that describes the situation of the dynamic object it is mapped to. A substate class is defined by a collection of predicate expressions: a substate belongs to a class if and only if it satisfies one of the expressions.*

In OCL_h developers specify all the legal substates that a typical object of a sort may occupy at the same time as developing the operator set. This helps in the understanding and debugging of the domain model, as well as contributing to the efficiency of planning tools. The specification is written implicitly as a list of predicate expressions such that any legal ground substitution of one of the expressions will be a hierarchical component of a substate. The legal substates of identifier *i* are thus all ground expressions having a component from exactly one of the predicate expressions at each level in the hierarchy.

The substate of Object in Example 2 actually has three hierarchical components - *at*, relating to *physical objects*, and *moveable, available*, relating specifically to *trucks*. Objects are described by predicates through their primitive sort (here -*truck*), but they also inherit the dynamic predicates from supersort (*physical_obj*).

```
(physical_obj, T, [[at(T,L)]]),
(truck, T, [[moveable(T),busy(T)]])
```

Example 2: Hierarchical substate specification for trucks

Definition 5 (Substate Transition Machine) *A substate transition machine is a Finite State Machine that describes the dynamics of a sort. Each node in the Finite State Machine (FSM) is annotated with a predicate expression defining a substate class. Each arc in the FSM represents a possible transition.*

Definition 6 (Invariants) *An* invariant *in OCL_h describes some set rules and facts to implement some constraints on the domain model in order to maintain, debug or to speed-up online planning.*

Invariants can be a) positive invariant ($Invs^+$) is an expression called *atomic invariants* which must be true in every planning state. A negative invariant ($Invs^-$) is an expression called *inconsistency constraints* which must be false in each planning state.

Definition 7 (Operator) *An operator term in OCL and OCL_h comprises an identifier and a list of parameters. There are three component to an operator: prevail conditions (conditions on substates that are true before and after execution of the operator), necessary effects (conditions that need to be true before the execution of the operator and are necessarily changed after it) and conditional effects (conditions on substates that if they were true before the operator executes, then they will be changed after it).*

An operator often know as *primitive operator* in OCL_h, has components (*Name, Prevail, Necessary, Conditionals*), in the syntax like:

operator($\mathscr{O}$ $(V_1,\ldots,V_n)$,
$[SSPrev_{V_1},\ldots]$
$[SSNecPre_{V_k} \Rightarrow SSNecPost_{V_k},\ldots]$
$[SSCondPre_{V_m} \Rightarrow SSCondPost_{V_m},\ldots]$)

Example 3: Syntax of primitive operator

$\mathscr{O}$ is the operator's *Name*, $(V_1,\ldots, V_n)$ is a set of parameters that their states are required to make the operator happens.

$[SSPrev_{V_1},\ldots]$ is called *Prevail* conditions. It is a set of the substates of objects that must be true before the operator can be executed and remain true during execution.

$[SSNecPre_{V_k} \Rightarrow SSNecPost_{V_k},\ldots]$ is called *Necessary* conditions. It is a set of necessary object transitions, $SSNecPre_{V_k}$ indicates the substate of object V_k before the task, while $SSNecPost_{V_k}$ indicates the substate afterwards.

$[SSCondPre_{V_m} \Rightarrow SSCondPost_{V_m},\ldots]$ is called *Conditionals* conditions. It is a set of conditional transitions, that if their exists an object V, its substates satisfy $SSCondPre_{V_m}$ before the operator is executed, its substates will then be changed to $SSCondPost_{V_m}$ after the execution of the operator. The object parameter in the *Conditionals* are therefore universally quantified over its sort, excluding any objects necessarily changed by the operator.

A primitive operator $\mathscr{O}$ can be applied to a state S if there is a grounding substitution t for *Necessary* and *Prevail* such that each transition in $Necessary_t$ can be applied to an object description in S, and each object expression in $Prevail_t$ is satisfied in S.

The new world state is S with

- the changes made to a set of objects as specified in the necessary transitions

- all *other* objects not affected by the necessary transitions, but which satisfy the *LHS* of a transition in *Conditionals*, changed according to that transition.

Example 4: shows a primitive operator of 'translog domain' specifying the movement of trucks between different cities.

```
Name:       move(V, O, L, R),
Prevail:    [ ],
Necessary:
            [sc(truck,V, [at(V,O),movable(V),in_city(O,City)] ⇒
            [at(V,L),in_city(L,City1),ne(City,City1),connects(R,City,City1),is_of_sort(R,road_route)])],
Conditionals:
            [sc(package,P,[loaded(P,V),at(P,O)]⇒ [loaded(P,V),at(P,L)])]
```

Example 5: Example of primitive operator

Definition 8 (Compound Operator) *A CompoundOperator in OCL_h is an action, which cannot be executed directly, requires further expansion until the primitive operators are found.*

In OCL_h, compound actions including, a) methods and, b) achieve(Goal) actions.

a) Methods

Methods are compound actions that can be expanded further down under certain restrictions. By eliminating the lower level tasks and orderings and variable binding that may lead to dead ends, the search space of a method expansion is greatly reduced.

The syntax of a method is as following:

Name: method($\mathscr{C}(V_1,\ldots, V_n)$),
Pre: $[SS_{V_1},\ldots]$
Transitions: $[SSPre_{V_k} \Rightarrow SSPost_{V_k},\ldots]$
Statics: $[ST_1,\ldots]$
Temps: [before(N_1,N_2),...]
Body: $[T_1,,\ldots,T_{N_1},\ldots,T_{N_2},\ldots]$)

Example 6:Syntax of a compound task (Method)

A **method** is defined as *(Name,Pre,Transitions,Statics,Temps,Bodies)*, that:

- *Name* of the method's name is $\mathscr{C}$ followed by its parameters $(V_1,\ldots, V_n)$
- *Pre* is a set of object substate expressions that must be true so that method $\mathscr{C}$ can be performed. Unlike the *prevail* in a primitive operator, substates $[SS_{V_1},\ldots]$ may be affected by the method while the expansion of $\mathscr{C}$.
- *Transitions* is a set of necessary state transitions similar to *Necessary* in the primitive operator case. However, because the method need to be expand to primitive operators, not like *necessary* in the primitive operators, *transitions* are the basic substates transitions known at top level. There may be more substates changes happens at the method expanding.

- *Statics* is the static constraints binding the parameters in the action. Again, the expansion of the method may bring more static constraints in later.
- *Temps* and *Body* are the restrictions of the expansion of the methods. *Body* defines the lower level tasks of its expanding. T_i can be a name of a primitive operator, or an *achieve(goal)* action (we will discuss it later) or a name of method that can be further expanded. *Temps* is the temporal orders of the lower level tasks. Here, number N_1 and N_2 refers to the N_1th and N_2th tasks in the *body* list – T_{N_1} and T_{N_2}. That T_{N_1} must be executed before T_{N_2}.

The following example shows a method for transport a package from one location to another location.

```
Name:      transport(P,O,D),
Pre:       [ ],
Transitions:
           [sc(package, P, [at(P,O)] ⇒ [at(P,D),delivered(P)])]
Statics:    [ne(O,D) ],
Temps:      [before(1,2), before(2,3)],
Body:
            [achieve(se(package, P,[waiting(P),certified(P)])),
            carry_direct(P,O,D),
            deliver(P,D)]
```

Example 5: Example of a method

In this example, the necessary substate transition of the method is package P must be changed from $[at(P,O)]$ to $[at(P,D]$, the method can only be applied when package P has the situation that it is $[at(P,O)]$. As defined by statics constraints, in this method, location O can't be the same as location D.

It can be expanded to actions: T_1*: achieve(se(package, P,[waiting(P), certified(P)]))*, T_2: *carry_direct(P,O,D)*, and T_3: *deliver(P,D)*, follow the order T_1 before T_2, and the T_3.

b) Achieve(Goal) action

In Example 5, we have an achieve(*Goal*) action
achieve(se(package, P,[waiting(P), certified(P)])).

Goal is a set of objects substates expressions that needs to be achieved by any actions. The expansion of an achieve(*Goal*) action is restrict by its precondition and *Goal* conditions. It can be expanded to any set of actions before we know its preconditions.

Definition 9 (*OCL* Planning Task) *Given a set of objects, invariants, ground initial substates, a set of goal substates and operators, an OCL planning task is to find a sequence of actions that each intermediate state satisfies all invariants and the final state satisfies the goal substates.*

3. Continuous Planning

Continuous planning deals with the planning in an environment where continuous changes may occur during the plan execution. In the PDDL+, continuous time is mod-

elled as real valued numbers. These numeric quantities are mainly dependent on the continuous processes. Planning in the continuous time environment is very complex. The well-known 'bath filling' domain is used as an example to express the continuous planning problem. An automated bath filling domain 'starts filling the bath tub in any pre-set time and it stops if it fills up to certain pre-set level'. The bath filling domain has been simulated by McCluskey [10] for continues process by using GIPO III's Plan Stepper. The procedure of the simulation for events and processes retracts all the stored nodes and each of them comes in turn. In the Plan stepper simulation (Figure 1), it is assumed that if there is more than one event triggers at the same time that must be independent to each other. Moreover, if there is an event triggers during process execution, it assumes not to interfere the process.

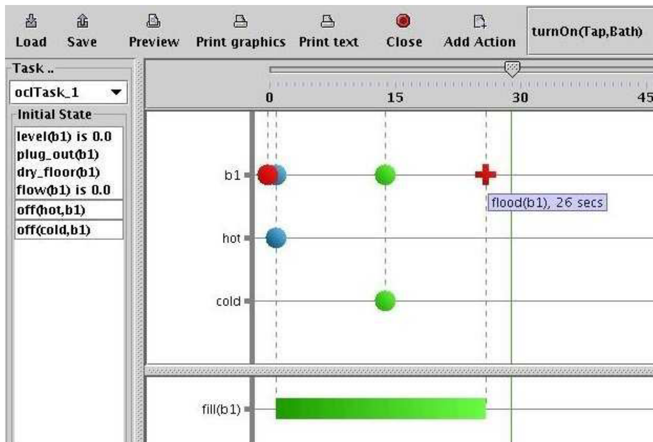

Figure 1. Bath filling domain simulation for continues time using GIPO III

In this extension of *OCLplus* we can represent the bath filling domain in terms of events and processes.

4. Continuous Planning in *OCLplus*

Continuous planning consists of processes which are responsible for either continuous or discrete change of object values (substates). A process is represented by a set of (recurrent) functions defined on a (local) time interval beginning at 0 (start of the process) modifying values of the objects (substates).

Definition 10 (Process) *A process is defined as a finite set of functions* $p = \{f_1^p, f_2^p, \ldots, f_n^p\}$ *such that each function* f_i^p *is defined as a mapping* $\mathscr{T} \rightarrow (M)_i^p$ *where* $\mathscr{T}$ *is a time interval and* $(M)_i^p$ *stands for a substate. Functions can be defined recurrently.*

We say that a function affects an object if and only if a range of values of the function is a corresponding substate of the object. We also say that a process affects an object if and only if one or more of its functions affects the object.

For example, we have a process *filling_tub* for filling the tub. The process contains only one function (representing a water level in the tub) which can be defined in the following way: $f(n) = f(0) + n \cdot V$ (V is a volume filled in one time unit). For emptying the tub (*empty_tub* process), we can have the similar function: $f'(n) = f'(0) - n \cdot V'$.

Processes can be executed or terminated by (*OCLplus*) operators or events. Operators (or actions which stands for ground instances of operators) can be applied by user (or agent) if their preconditions are met. Events, on the other hand, are triggered automatically (without user's or agent's interference) if their preconditions are met. Deterministic events are triggered always while non-deterministic events might be triggered if their preconditions are met.

Definition 11 (*OCLplus* Operator) *An OCLplus operator o is a tuple* $o = (pre(o), start(o), stop(o))$ *where* $pre(o)$*, a precondition of o, is a set of expressions which must be true before applying o,* $start(o)$ *is a set of processes which start after o is applied (if not already running) and* $stop(o)$ *is a set of processes which are stopped after o is applied.*

Definition 12 (Event) *An event e is a tuple* $e = (pre(e), start(e), stop(e))$ *where* $pre(e), start(e)$ *and* $stop(e)$ *are defined in the same way as in OCLplus operator case. If an event e is deterministic then it is triggered always whenever its precondition* $(pre(e)$ *is met. If an event e is non-deterministic then if its precondition* $(pre(e))$ *is met then it may be triggered.*

Considering our *bath* example, we can define an operator *start_filling* (with empty precondition) which executes the process *filling_tub*. We can also define an operator *stop_filling* having a precondition that water level in the tub must be at least l and which stops the process *filling_tub*. If the tub has an overflow drain, then we can define a deterministic event *over_flow* which triggers the process *empty_tub* when the water level reaches a critical value l_c. Similarly, we can define an event *stop_over_flow* which triggers when the water level falls below the critical value.

However, it might happen that operators or events with contradictory effects (e.g. some operator or event is going to execute a process while some other operator or event is going to stop it) can be executed at the same time. Therefore, it must be explicitly specified (in the problem definition) which effect has a priority.

Processes which are only responsible for modifying object values work with local information, i.e., a process does not know whether any other process affects values of the same objects. Therefore, we have to keep a global information about the actual object values. For this purpose we have to introduce a special object representing global time c_t and a special process p_t representing a global timer. The global times starts at the beginning of the planning process and no operator or event can stop it. For an object c it must hold the following ($c(t)$ refers to value of c in time t):

- **Prevailing:** if no running process affects the value of c in an open time interval (t, t'), then $\forall x \in (t, t') : c(x) = c(t)$
- **Exclusivity:** if there is only one running process and only one of its function f affects the value of c in an open time interval (t, t') (the process starts in the time t and ends in the time t'), then $\forall x \in (t, t') : c(x) = f(x - t)$.

- **Simultaneous affecting:** if a set of the functions $\{f_1, \ldots, f_k\}$ defined in the running processes affect the value of c in an open time interval (t,t') and all the functions from the set are in the recurrent form (i.e. $f(n) = f(0) + g(n)$), then $\forall x \in (t,t') : c(x) = c(t) + \sum g_i(t'-t)$ (note that $f_1(0) = f_2(0) = \ldots = f_k(0) = c(t)$).

First two conditions are straightforward to follow because it is obvious that value of the object cannot be modified if no running process affect it or if just one function (in the running processes) affect the value of the object then it is modified directly according to the function. If more functions affect the same object in the same time then the functions must be in defined in the recurrent form which means, informally said, that such a function only increases or decreases the value of the object. In our example, if the processes *filling_tub* and *empty_tub* run simultaneously, then both affect the water level in such a way that in n units of time it is changed by $nV - nV'$.

However, if functions (in the running processes) modify the value of the object by assignment regardless of object's previous value (e.g. $f(n) = x$), then it may cause inconsistency (e.g. the object cannot have two different values at the same time).

An (*OCLplus*) Planning Task is defined via sets of objects, substates, processes, (*OCLplus*) operators, Events and initial and goal situations. An initial situation gives all the objects initial values (e.g. the water level is 2). A goal situation is defined by a set of expressions (e.g. the water level is greater than 5).

Definition 13 (*OCLplus* Planning Task) *An OCL Plus Planning Task is a tuple* $\Pi = (O,S,\mathscr{P},\mathscr{O},\mathscr{E},I,G)$ *where O is a set of objects, S in a set of substates,* $\mathscr{P}$ *is a set of processes,* $\mathscr{O}$ *is a set of OCL Plus operators,* $\mathscr{E}$ *is a set of events, I is a set of initial ground substates and G is a set of goal expressions.*

Since we deal with planning, we have to somehow represent a plan. In contrast to classical planning, where a plan is a sequence of actions, here we have to consider also events and time-stamps in which an action (an instance of an operator) or event was executed.

Definition 14 (Plan) *A plan* π *is a sequence of pairs* (a_i,t_i) *or* (e_j,t_j) *where* (a_i,t_i) *denotes an action* a_i *executed in time* t_i *and* (e_j,t_j) *denotes an event* e_j *triggered in time* t_j*.*

A plan is valid if all action/event-time-stamp pairs follows the following conditions:

- for all action-time-stamp pairs (a_i,t_i) it holds that $pre(a_i)$ is met in time t_i
- for all event-time-stamp pairs (e_i,t_i) it holds that $pre(e_i)$ is met in time t_i
- if $pre(e_i)$ is met in time t_i and e_i is a deterministic event, then (e_i,t_i) must be in the plan

A plan is a solution of (*OCLplus*) planning task if the plan is valid and all goal expressions are satisfied at some point.

In our simple 'bath filling' example, where in the initial situation the water level is 2 and we want to increase it to 5 (the goal situation), then we simply execute the action *start_filling* in time 0. It starts the process *filling_tub* which eventually causes increasing the water level to the desired value.

5. Future Work

This work is the start of a larger programme of study in developing *OCLplus*. There are semantic issues relating to concurrency for which there remain unresolved questions. For example, given two events that trigger with the same precondition but which have conflicting effects, which of them is executed (or is the model invalid). Many similar issues to this one are resolved in particular ways in the PDDL+ language. However, given the multi-valued nature (the state machines that represent object states can be seen as finite domain variables) of *OCLplus*, we will reconsider these issues as it could be possible that alternative semantics are more appropriate.

Also, we wish to allow a greater degree of syntactic expression in *OCLplus* than is available in PDDL+. The value of the global (and local process) clock(s) will be available as a fluent to precondition on (but of course never to explicitly change) and to use on the right hand side of assignments. One benefit to this is that it will be far simpler to construct goals such as minimising the sum tardiness of several jobs. Although the authors have not demonstrated this, it is likely that this is possible to achieve the same result in pure PDDL+ using dummy actions or processes. However, if this were possible in the core language, planning algorithms could exploit this feature.

Clearly, a modelling language in isolation is not our final goal: we do intend to continue development on both hierarchical and non-hierarchical planning systems. Clearly, solving planning problems with continuous time and exogenous events is a difficult task, but we envisage that some features inherent in *OCL* provide promising avenues in gaining leverage here. We expect that one benefit to using a multi-valued representation will be the exploitation of similar structures as Domain Transition Graphs(DTG) and Causal Graphs studied in the context of SAS+. For example, analysis of the DTG of an object may reveal which processes are required to solve an *OCLplus* problem. Modelling is also an important issue and we will develop a greater range of domains to experiment on.

6. Conclusions

This paper presents an extension of a well-known language *OCL*, earlier used in GIPO to model planning domains, to *OCLplus* that allows modelling also features characteristic for continuous planning (e.g. processes). Processes are responsible for continuous or discrete changes of object values. Actions, executed by a user, and events, executed automatically, are responsible for executing and terminating processes. *OCLplus*, therefore, enables modelling of continuous planning tasks, however, some issues need to be addressed in future as discussed before. For instance, if actions or events executed in the same time have conflicting effects. Introducing *OCLplus* is obviously not a final goal. Our plans for future work consist of developing planning systems exploiting advantages of object centred modellling. Moreover, the planing systems should support hierarchisation (such as in HTNs) which allows to solve more complex (real-world) problems.

References

[1] John Bresina, Nicolas Meuleauy, Sailesh Ramakrishnan, David Smith, and Rich Washingtonx. Planning under continuous time and resource uncertainty: A challenge for ai. In *In Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, pages 77–84. Morgan Kaufmann, 2002.

[2] L.P. Carloni, R. Passerore, A. Pinto, and A. Sangiovanni-Vincentelli. *Languages and tools for hybrid systems design*. 2006.

[3] Patrick Eyerich, Thomas Keller, Bernhard Nebel, and Albert ludwigs-universitt Freiburg. Combining action and motion planning via semantic attachments. In *International Conference on Automated Planning and Scheduling*, 2010.

[4] J. Fdez-Olivares, L. Castillo, O. Garcia-Perez, and F. P. Reins. Bringing users and planning technology together: experiences in SIADEX. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006)*, pages 11 – 20, Cumbria, UK, 2006.

[5] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains . In *Technical Report, Dept of Computer Science, University of Durham*, 2001.

[6] M. Fox and D. Long. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27:235 – 297, 2006.

[7] Maria Fox, Derek Long, and Daniele Magazzeni. Automatic construction of efficient multiple battery usage policies. In *International Conference on Automated Planning and Scheduling*, 2011.

[8] T. L. McCluskey, D. Liu, and R. M. Simpson. GIPO II: HTN Planning in a Tool-supported Knowledge Engineering Environment. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling*, pages 92 – 101. AAAI Press, Menlo Park, California, 2003.

[9] T. L. McCluskey and J. M. Porteous. Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency. *Artificial Intelligence*, 95:1–65, 1997.

[10] T.L. McCluskey and R.M. Simpson. Tool support for planning and plan analysis within domains enbodying continuous change. In *Workshop on Plan Analysis and Management held in conjunction with The 16th International Conference on Automated Planning and Scheduling, (ICAPS 2006)*, June 2006.

[11] C. McGann. How to solve it: Problem solving in Europa 2.0. Technical report, NASA Ames Research Centre, 2006.

[12] N. Muscettola. HSTS: Integrating planning and scheduling. In *Intelligent Scheduling*, pages 169–212. Morgan Kaufmann, 1994.

[13] A. Riabov and Z. Liu. Scalable planning for distributed stream processing systems. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, Cumbria, UK, 2006.