A Visual Integrated Development Environment for Automated Planning Domain Models

James C. Ong[1] and Emilio Remolina[2]
Stottler Henke Associates, Inc., San Mateo, CA, 94402

David E. Smith[3]
NASA Ames Research Center, Moffet Field, CA, 94035

and

Mark S. Boddy[4]
Adventium Labs, Inc. Minneapolis, MN, 55401

Automated planning software uses symbolic reasoning techniques and models of the planning domain to generate plans. Execution systems execute these plans to perform tasks or achieve goals, subject to constraints imposed by physical laws, resource limits, and environmental conditions and flight rules. Automated planning can support autonomous operations of spacecraft, habitats, and space launch systems. The Action Notation Modeling Language (ANML) is a relatively new language developed by NASA for specifying planning domain models. Developing and maintaining good planning domain models is challenging and critical to the success of applying applying automated planning technology to support autonomous systems. We developed an integrated development environment (IDE) to help modelers enter, review, test, debug, maintain, and enhance ANML planning domain models as well as review and understand ANML models developed by others. The IDE is implemented in the Java programming language using the Eclipse and Xtext open source frameworks for developing integrated development environments (IDEs). The IDE provides a syntax-aware text-based editor that color-codes ANML text based on its syntactic type, flags errors and warnings, supports browsing, suggests code completions, and provides on-line help. It automatically detects and highlights problems such as syntax errors, type mis-matches, references to undefined variables, and incorrect numbers or types of arguments in references to variables and actions. In addition, the IDE provides graphical displays that help modelers see important patterns and relationships among planning variables and actions. An evaluation showed that PM/IDE significantly reduced the time needed to create ANML models. The syntax highlighting and tooltips helped modelers avoid many syntax errors, and the visualizations helped modelers identify logic errors and other semantic issues.

[1] Group Manager, 1670 S. Amphlett Blvd, Suite 310, San Mateo, CA 94402, Associate Member.
[2] AI Project Manager, 1670 S. Amphlett Blvd, Suite 310, San Mateo, CA 94402.
[3] Computer Scientist, Intelligent Systems Division, M/S 269-2, Moffett Field, CA 94035.
[4] Chief Scientist, 111 Third Ave South, Suite 100, Minneapolis, MN 55401.

I. Motivation

Automated planning software uses symbolic reasoning techniques and models of the planning domain to generate plans. Execution systems execute these plans to perform tasks or achieve goals, subject to constraints imposed by physical laws, resource limits, and environmental conditions. Automated planning can support autonomous operations of spacecraft, habitats, and space launch systems.

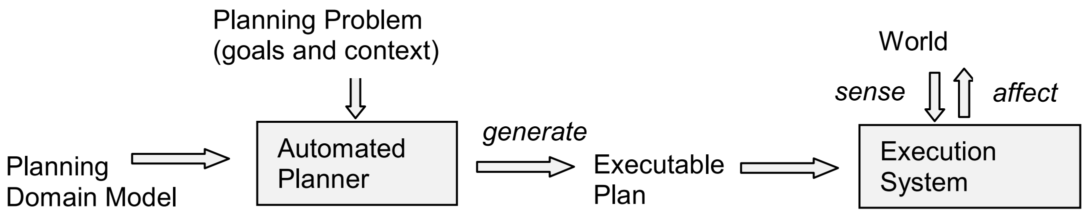


Figure 1. Model-based plan generation and execution.

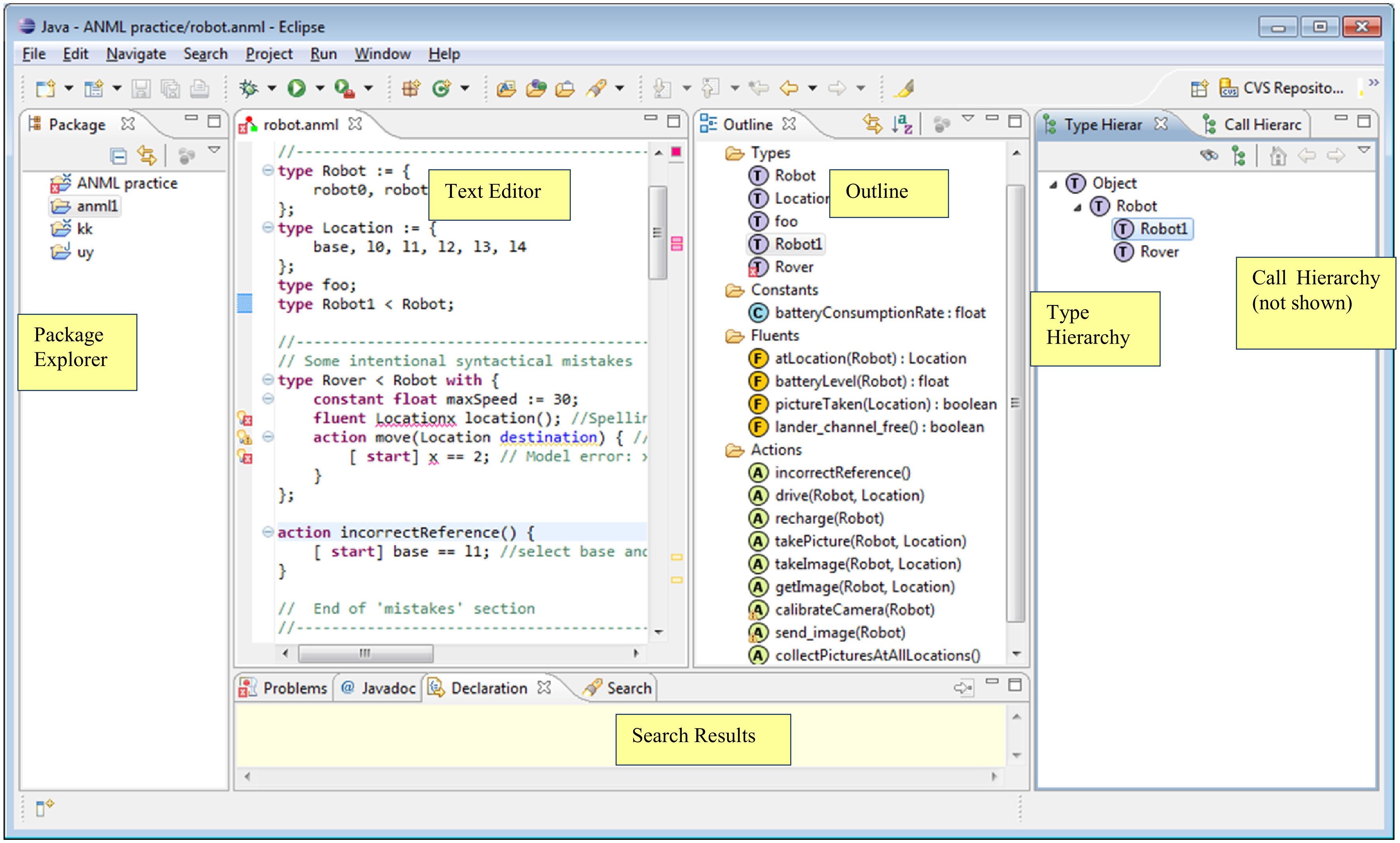
A good domain model is sufficiently complete and correct, understandable, enables the automated planner to generate efficient plans reasonably quickly, is easily maintainable, and enables the implications of the model to be seen easily. Developing planning domain models is challenging because it requires the modeler to encode abstractions of the world state, possible actions, and their temporal and non-temporal relationships as statements in a specialized modeling language. In addition, modelers must notice important, sometimes subtle interactions among different parts of the model and understand how they affect the range of plans which can and cannot be generated. The feasibility of applying automated planning technology is strongly affected by the difficulty and effort required to develop good planning domain models.

II. ANML Language Overview

The Action Notation Modeling Language (ANML)[1] is a relatively new language developed by NASA for specifying planning domain models. ANML's design is inspired by features of the Planning Domain Definition Language (PDDL) [2] commonly used by the planning research community, the New Domain Definition Language (NDDL) used with the EUROPA system at NASA[3], and the Aspen Modeling Language (AML) used with Aspen at JPL[4]. One describes the world in ANML by specifying the state of the world and the actions that act upon the world. The state of the world is represented by discrete and continuous fluents which are variables and functions whose values can change during the planning horizon. All fluents must be typed, and complex types can be declared in terms of more primitive types. ANML allows the declaration of structured types whose instances have predefined properties. This provides a convenient and clear way of defining fluents and actions and organizing them around the types of objects they describe. Actions in ANML have quantitative durations. ANML permits rich temporal constraints on action conditions and effects. For example, one can specify that conditions must hold at any time or during any time interval during an action, and effects can take place at any time, not just at the start or end of durative actions. ANML contains convenient idioms for expressing common types of resource usage, and it contains mechanisms for describing hierarchical task decomposition. ANML allows the specification of temporal constraints on goals using the same primitives as are used in the specification of rich constraints on action conditions. ANML allows the specification of exogenous conditions (facts) that hold at times other than just the start of the planning horizon.

III. Planning Model Integrated Development Environment

We designed the Planning Model Integrated Development Environment (PM/IDE) to help users create, review, understand, and project the effects of ANML planning domain models more quickly, easily, and effectively. PM/IDE is implemented as an Eclipse plug-in that helps users edit, analyze, and debug planning domain models expressed in the Action Notation Modeling Language (ANML). PM/IDE relies on the Xtext open source framework for developing domain-specific languages within Eclipse. PM/IDE provides a collection of Eclipse views, including a syntax-aware ANML Text Editor. This view color-codes ANML text based on its syntactic type, flags errors and warnings, supports querying and browsing, and suggests code completions. PM/IDE differs from previously-developed planning domain modeling tools [5, 6,7,8,9,10,11,12,13,14,15,16] in its heavy use of visualization techniques and its focus on the ANML language.


Java - ANML practice/robot.anml - Eclipse
File Edit Navigate Search Project Run Window Help
CVS Reposito...
Package
ANML practice
anml1
kk
uy
robot.anml
Outline
Types
Robot
Locatior
foo
Robot1
Rover
Constants
batteryConsumptionRate : float
Fluents
atLocation(Robot) : Location
batteryLevel(Robot) : float
pictureTaken(Location) : boolean
lander_channel_free() : boolean
Actions
incorrectReference()
drive(Robot, Location)
recharge(Robot)
takePicture(Robot, Location)
takeImage(Robot, Location)
getImage(Robot, Location)
calibrateCamera(Robot)
send_image(Robot)
collectPicturesAtAllLocations()
Type Hierar
Call Hierarc
Object
Robot
Robot1
Rover
Problems
Javadoc
Declaration
Search
Text Editor
Outline
Package Explorer
Type Hierarchy
Call Hierarchy (not shown)
Search Results

```
//---------------------------------------
type Robot := {
    robot0, robot
};
type Location := {
    base, l0, l1, l2, l3, l4
};
type foo;
type Robot1 < Robot;

//---------------------------------------
// Some intentional syntactical mistakes
type Rover < Robot with {
    constant float maxSpeed := 30;
    fluent Locationx location(); //Spellir
    action move(Location destination) { //
        [ start] x == 2; // Model error: >
    }
};

action incorrectReference() {
    [ start] base == l1; //select base anc
}

//  End of 'mistakes' section
//---------------------------------------
```

Figure 2. PM/IDE text views from left to right: Package Explorer, ANML syntax-aware text editor, outline, and type hierarchy.

Figure 2 shows the views provided by the PM/IDE Eclipse plug-in:

- **Package Explorer** – displays the names of ANML files the current ANML project.
- **Text Editor** – displays the ANML file selected in the Package Explorer. Each word is color-coded, based on its syntactic type. The Text Editor flags errors and warnings, supports querying and browsing, and suggests code completions.
- **Outline** – displays a catalog of actions, types, constants and fluents defined in the ANML file displayed in the Text Editor. Double-click on an element in the Outline to display its declaration or definition in the Text Editor.
- **Type Hierarchy** – displays the supertypes and subtypes of a user-selected type.
- **Call Hierarchy** – displays the hierarchy of actions that include the selected action as a subaction in a decomposition.
- **Search Results** – displays the results of the *Find References, Find Writers,* and *Find Readers* operations invoked from the context menu.

PM/IDE also provides visualizations that help users see relationships among actions and fluents. The **Action Fluents Timeline Summary** shows when a user-selected action reads, writes, or constrains the action's parameters and local and global fluents. An action reads a fluent or parameter if the fluent or parameter appears on the right side of an assignment statement. An action writes a fluent if the fluent appears on the left side of an assignment statement. An action constrains a fluent if the fluent appears in a Boolean condition. An action increments or decrements a fluent if it uses the ANML increment or decrement operator to increase (or decrease) its value by an amount. The value of a fluent can be specified to be undetermined at a point in time or over a time interval. A red circle or horizontal bar indicates an invalid specification of a fluent condition or assignment.

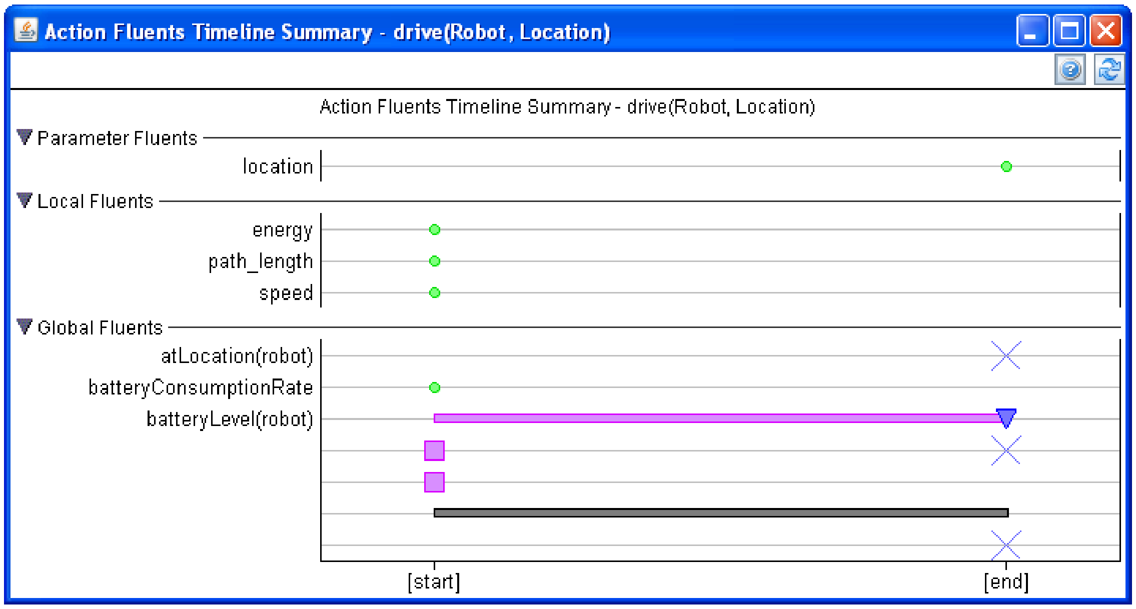
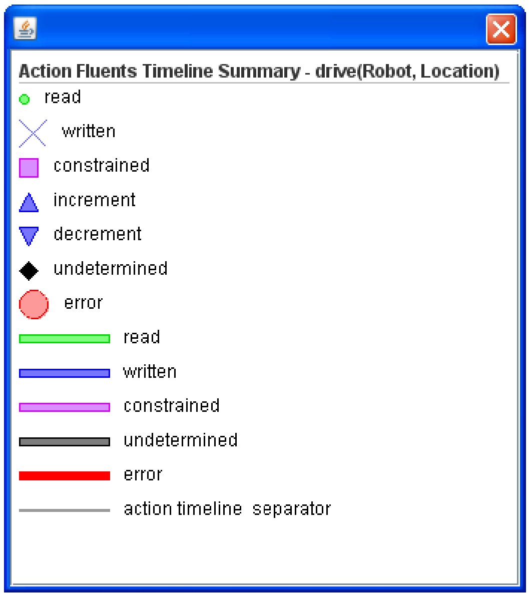


Figure 3. Action Fluents Timeline Summary.

This view contains three sections labeled *Parameter Fluents*, *Local Fluents*, and *Global Fluents*. In each section, there is one timeline for each reference to a fluent or parameter in the action. A symbol represents a reference to a fluent or parameter at a point in time such as [start], [end], or [start+5]. A horizontal bar represents a reference that spans a time interval, such as [start+5, end].

Figure 3 shows a number of semantic problems in the definition of the drive(Robot, Location) action. First, the row labeled atLocation(robot) shows that the location is updated at the end of the action. However, the location should be undetermined during the drive action. The last five rows labelled batteryLevel(robot) represent five references to this fluent within the action definition, and some of them are incompatible. For example, the first row shows a decrement at the end of the action, and the second and fifth rows show that the fluent is written, and only one of these three can be included in a valid action.

The **Fluent Actions Timeline Summary** shows when a user-selected fluent is read, written, or constrained by any action in the model. The horizontal positions of the symbols and horizontal bars indicate the timing of each fluent relative to the start and end time of the particular action in which each fluent is referenced.


Action Fluents Timeline Summary - drive(Robot, Location)

●	read
✕	written
■	constrained
▲	increment
▼	decrement
◆	undetermined
●	error
🟩	read
🟦	written
🟪	constrained
⬛	undetermined
🟥	error
—	action timeline separator

Figure 4. Action Fluents Timeline Summary key.

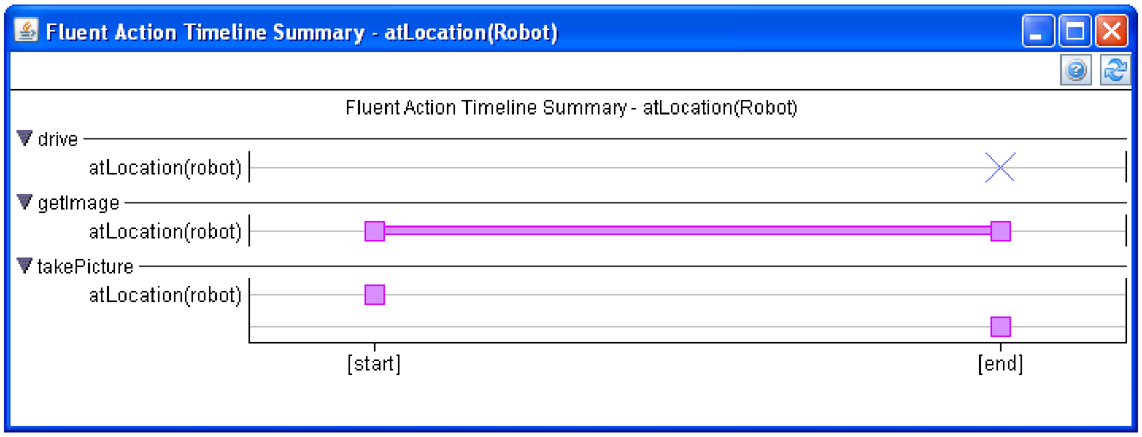
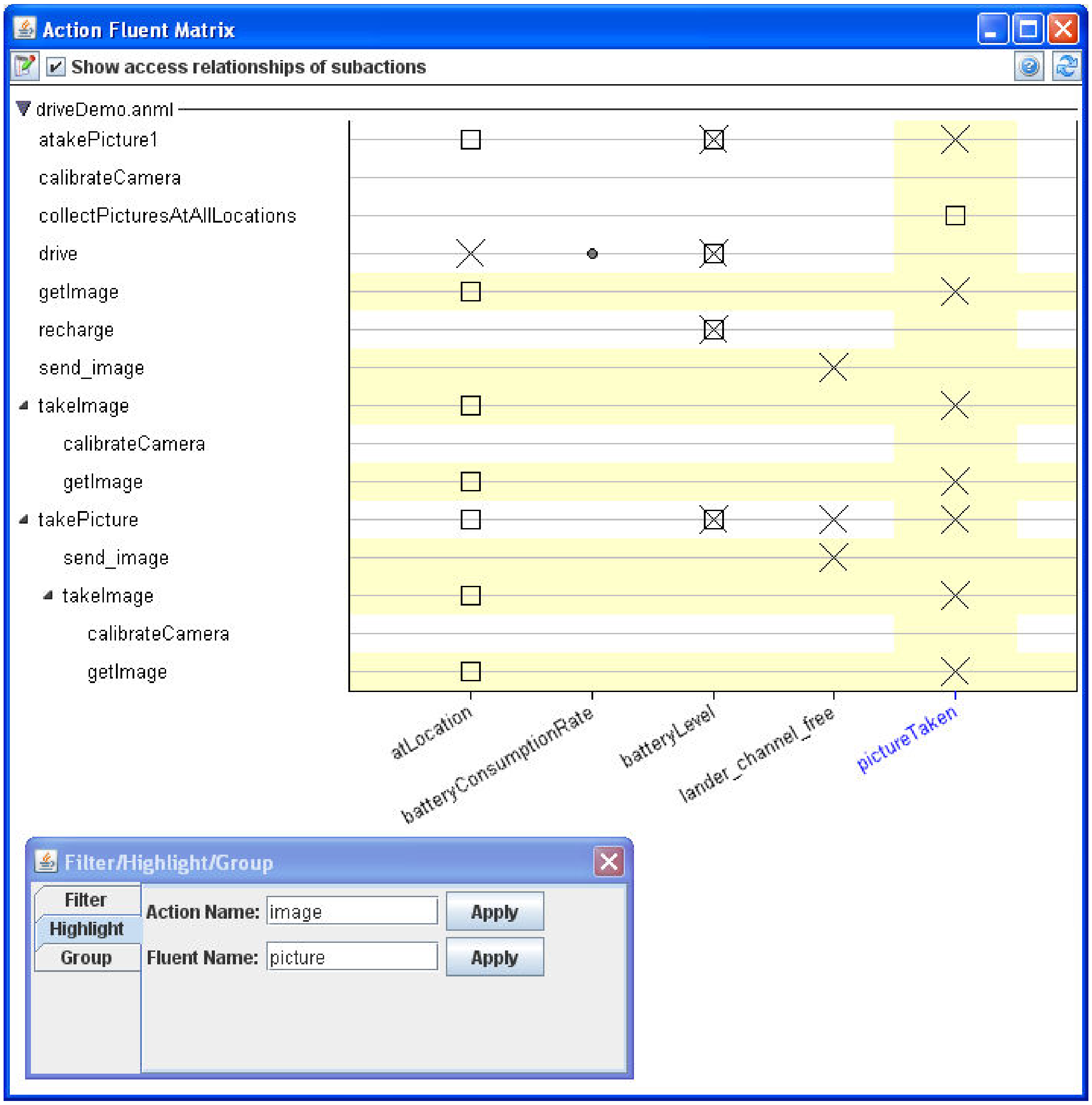

Fluent Action Timeline Summary - atLocation(Robot)
Fluent Action Timeline Summary - atLocation(Robot)
drive
atLocation(robot)
getImage
atLocation(robot)
takePicture
atLocation(robot)
[start]
[end]

Figure 5 – Fluent Actions Timeline Summary.

The **Action Fluent Matrix** shows relationships between actions and global fluents. This display contains one row per action and one column per global fluent. At each row-column position, up to three overlapping symbols are drawn to indicate whether the action reads, writes, or constrains the fluent. Grouping, filtering, and highlighting features help users see relationships in large models that contain many actions and fluents.


Action Fluent Matrix

[✓] Show access relationships of subactions

driveDemo.anml

	atLocation	batteryConsumptionRate	batteryLevel	lander_channel_free	pictureTaken
atakePicture1	□		⊠		✕
calibrateCamera					
collectPicturesAtAllLocations					□
drive	✕	●	⊠		
getImage	□				✕
recharge			⊠		
send_image				✕	
takeImage	□				✕
calibrateCamera					
getImage	□				✕
takePicture	□		⊠	✕	✕
send_image				✕	
takeImage	□				✕
calibrateCamera					
getImage	□				✕

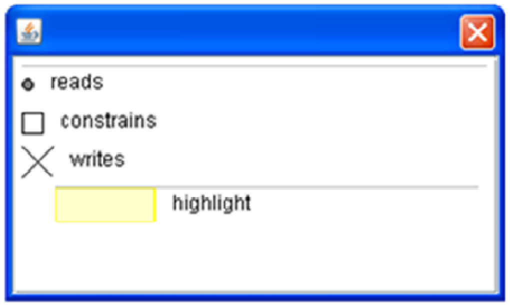
Filter/Highlight/Group

Filter | Highlight | Group

Action Name: image [Apply]

Fluent Name: picture [Apply]

Figure 6. Action Fluent Matrix, rows, and columns corresponding to user-selected actions and/or fluents are highlighted.


● reads

□ constrains

✕ writes

🟨 highlight

Figure 7. Action Fluent Matrix graph key.

The **Action Dependency Matrix** displays two kinds of relationships between pairs of actions: (1) *Upstream/Downstream* and (2) *Accesses Same Fluents*. An action is upstream of another action if the upstream action writes a global fluent that is read or constrained by the downstream action. On the other hand, an action accesses the same fluents as another action if the two actions access (read, write, or constrain) any fluents in common. The presence of a blue square at a row and column indicates that the action for that row (A) is *directly related* to the action for the column (B). For example, in Figure 8, action *drive* is immediately upstream of action *atakePicture1*. The presence of a light-blue square indicates that the action for that row (A) is *indirectly related* to the action for that column (C). For example, in Figure 8, action *drive* is upstream of action *collectPicturesAtAllLocations,* but it is not immediately upstream, so the square is light blue. Filtering and highlighting features help users see relationships in large models containing many actions.

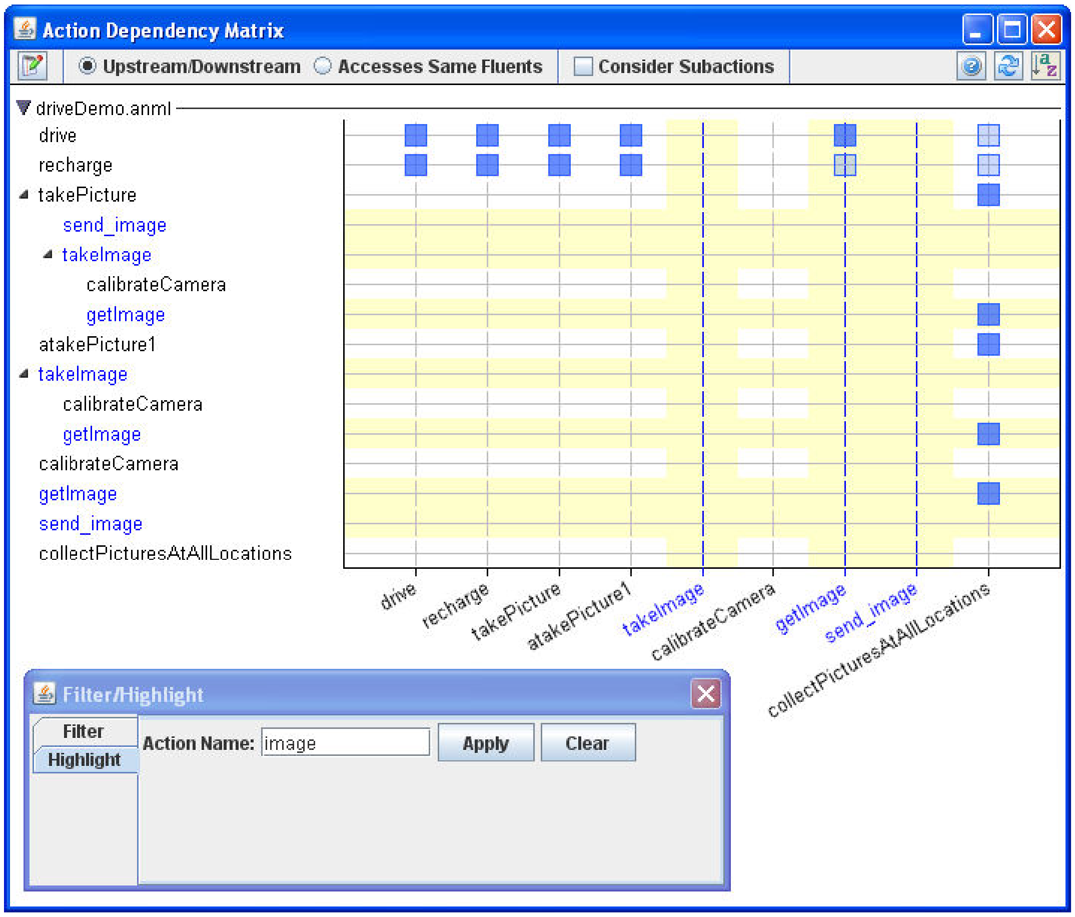
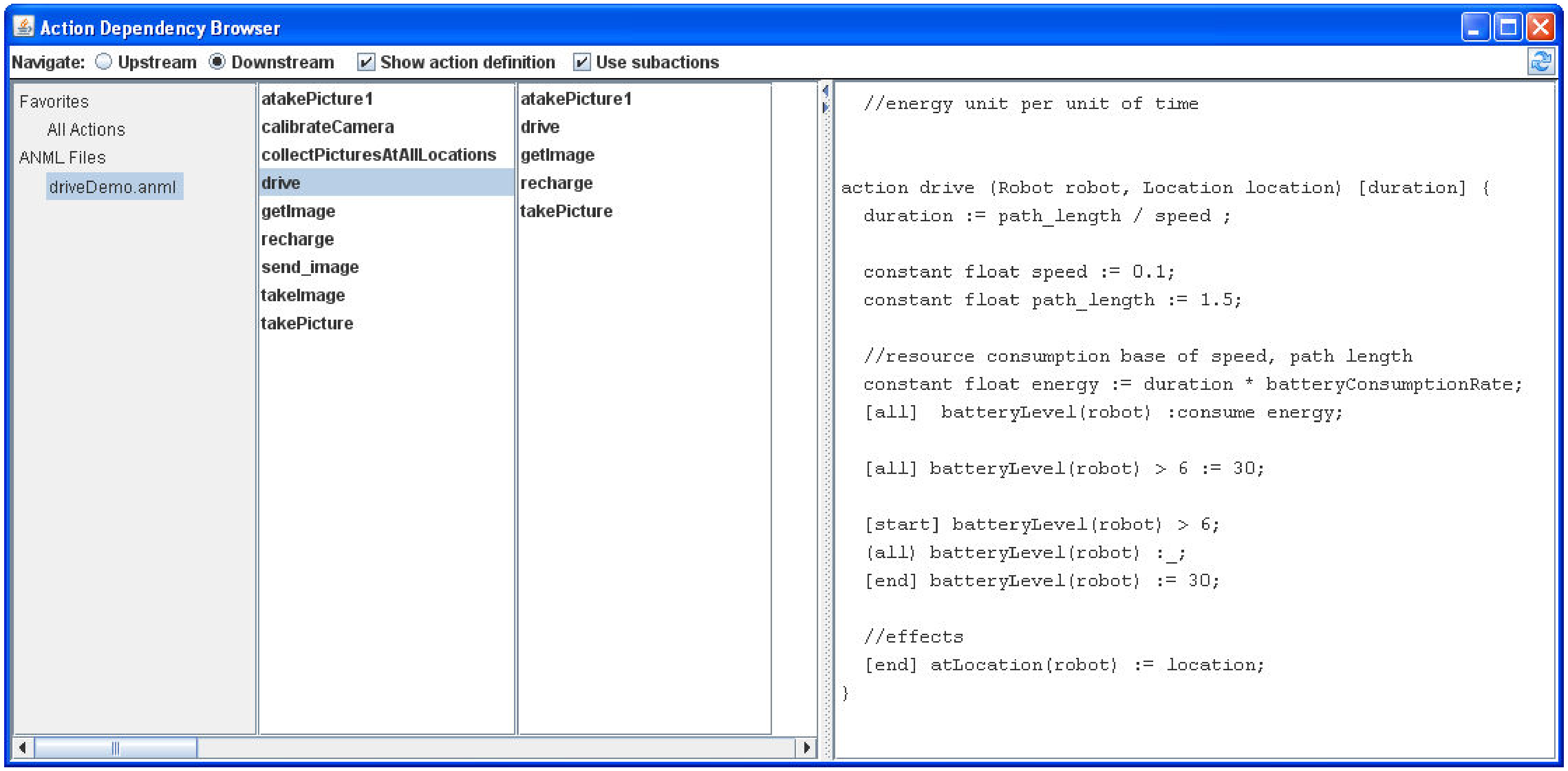


Figure 8. Action Dependency Matrix.

The **Action Dependency Browser** enables users to browse actions by following upstream or downstream relationships between actions. An action A is *upstream* of action B if action A writes a fluent that is read or constrained by action B. Conversely, an action B is *downstream* of action A if action B reads or constrains a fluent that is written by action A. The *Action Dependency Browser* is similar to the Macintosh Finder. The leftmost gray column displays the names of collections of actions. For example, users can click on *All Actions* to display the names of all actions in the model in the second column of the Browser. Or, users can click on the name of an ANML file in the leftmost column to display the names of actions defined in that file in the second column, as shown in Figure 9. The Browser can be configured to follow either upstream or downstream relationships between actions. If the *Downstream* radio button has been selected, when the user clicks to select an action (A) in a column (C), the column to the right of column C is refreshed to display actions that are downstream of action A. For example, in Figure 9, action *drive* is selected in the second column, so the third column lists actions that are downstream of action *drive*. If the user clicks on an action in the third column, PM/IDE will display immediately downstream actions in a fourth column.



```
  //energy unit per unit of time


action drive (Robot robot, Location location) [duration] {
  duration := path_length / speed ;

  constant float speed := 0.1;
  constant float path_length := 1.5;

  //resource consumption base of speed, path length
  constant float energy := duration * batteryConsumptionRate;
  [all]  batteryLevel(robot) :consume energy;

  [all] batteryLevel(robot) > 6 := 30;

  [start] batteryLevel(robot) > 6;
  (all) batteryLevel(robot) :_;
  [end] batteryLevel(robot) := 30;

  //effects
  [end] atLocation(robot) := location;
}
```

Figure 9. The third column of the Action Dependency Browser shows actions that are downstream of the drive action.

IV. Evaluation Design

Adventium Labs evaluated the strengths and weaknesses of PM/IDE as a tool for developing and maintaining planning domain models, specifically in ANML. In particular, Adventium Labs was interested in the degree to which the PM/IDE would help or hinder the development process in the hands of a moderately experienced modeler who did not happen to be expert at using the PM/IDE. The evaluation was limited to installation and use of the syntax-aware ANML Text Editor, supporting views, and the visualizations used to view relationships between actions and fluents.

During the evaluation, two modelers were tasked with developing an ANML model of International Space Station extra-vehicular activities (EVAs) using good design principles and a wide range of ANML features. They were provided with an incomplete PDDL-e model and access to a subject matter expert. PDDL-e does not include the same kind of typing system, such as structured types, that are available in ANML. Also, ANML also enables temporal constraints to be expressed much more flexibly and powerfully. Thus, the differences between the ANML and PDDL-e languages, combined with their instructions to use ANML features, discouraged them from simply translating the PDDL-e model into ANML in a literal way.

The choice to use this model was driven by a combination of factors. First, one of the features of both ANML and PM/IDE to be exercised and evaluated was the presence of task decomposition (action/sub-action hierarchy). There is a wide range of well-described planning models available through the International Planning Competition (IPC). However, none of them are specified as action hierarchies, so none were viewed as suitable for this evaluation.

The domain was comprised of eight sub-domains, 391 types, 243 actions, and 340 predicates. One modeler generated a portion of the EVA model in ANML using the PM/IDE, tracking time elapsed, errors found, and impressions regarding the help or hindrance provided by different features of the PM/IDE. A second modeler generated the same portion of the EVA model in ANML using the Emacs text editor and checked syntax using an ANML parser. The models were then compared for length and structure. Finally, the second modeler's hand-generated model was debugged using both PM/IDE and a combination of Emacs and the parser, this time with type-checking enabled. The purpose of the final step was to provide a quick evaluation of PM/IDE's capabilities for debugging and maintaining a model that had already been developed.

V. Evaluation Results

PM/IDE significantly reduced the time needed to develop ANML models. The same segment of code that took one developer approximately 15 hours while using the PM/IDE took the second developer approximately 22 hours without it. Use of PM/IDE had no significant effect on the structure or length of the planning model developed. Both modelers, sharing a moderate knowledge of planning and the ANML language but strong software engineering backgrounds, ended up with very similar models in terms of the number of lines, types, fluents, and actions. Both modelers took advantage of the object-oriented aspects of ANML as compared to the PDDL-e version.

The syntax highlighting and tooltips helped avoid hundreds if not thousands of trivial syntax errors (e.g., missing semicolons or illegal names) that would have taken much longer to correct without access to a syntax-aware editor. Evaluating the model under development using the visualizations (the Action Fluents Timeline Summary and Action Fluents Matrix in particular) helped to identify logic errors (that were syntactically correct) and other semantic issues in the model that would not have been discovered until the planner generated an unexpected plan (and maybe not discovered at all) if the only tools available had been a basic text editor and parser. The model developed for this evaluation was not especially large or complex; the value of the PM/IDE will be even more evident with larger models.

During the evaluation, there were minor bugs in the IDE that need to be worked out, and these bugs slowed the modeling process. Nearly half the time spent on this evaluation was expended tracking down bugs in the PM/IDE, documenting them, and revising the model to accommodate them. If this time had been excluded, fewer than 10 hours would have been required to develop the model using PM/IDE. There were some inconsistencies between correct syntax as described in the ANML manual and the PM/IDE-accepted syntax. The grammar of the ANML language is still changing, so PM/IDE should identify which language specification it supports. After the evaluation, Stottler Henke repaired most of the reported software bugs. However, some limitations remain. In particular, PM/IDE does not recognize relationships between actions and fluents that are fields within structured objects. Also, it does not support object type inheritance.

VI. Conclusions

The syntax-aware text editor and visualizations provided by PM/IDE enable modelers to create planning domain models expressed in the ANML language more quickly, reducing the effort needed to incorporate automated planning technology within autonomous spacecraft, habitats, and space launch systems.

Acknowledgments

Funding for this research was provided by NASA contract NNX11CC18C.

References

[1] Smith, D., J. Frank, W. Cushing (2008). The ANML Language. *ICAPS-08 Poster Session,* Sydney, Australia, Sept. 14-18, 2008.

[2] Fox, M., & Long, D. (2003). PDDL2.1: An extension of PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)*, 20, 61–124.

[3] Frank, J., & J´onsson, A. K. (2003). Constraint-Based Attribute and Interval Planning. *Journal of Constraints Special Issue on Constraints and Planning,* 8(4), 339–364.

[4] Sherwood, R.; Engelhardt, B.; Rabideau, G.; Chien, S.; and Knight, R. 2005. *ASPEN User's Guide.* Technical Report D-15482, JPL.

[5] Daley, P., J. Frank, M. Iatauro, C. McGann, W. Taylor (2005). PlanWorks: A Debugging Environment for Constraint-Based Planning Systems. *ICAPS 2005 Competition on Knowledge Engineering for Planning and Scheduling.* Monterey, California, USA.

[6] Edelkamp, S., & Mehler, T. (2005). Knowledge acquisition and knowledge engineering in the ModPlan workbench. In *Proceedings of the 1st International Competition on Knowledge Engineering for AI Planning*, Monterey, California, USA.

[7] Myers, K. (2007) Temporal Summarization of Plans. *Proceedings of the ICAPS-2007 Workshop on Moving Planning and Scheduling Systems into the Real World.* Providence, Rhode Island.

[8] McCluskey, T. L., & Simpson, R. M. (2006). Tool Support for Planning and Plan Analysis within Domains Embodying Continuous Change. In *Proceedings of the ICAPS 2006 Workshop on Plan Analysis and Management.* Cumbria, UK.

[9] McCluskey, T., Cresswell, S., Richardson, N., & West, M. M. (2010). Action Knowledge Acquisition with Opmaker2. In Filipe, J., Fred, A., & Sharp, B. (Eds.), Agents and Artificial Intelligence, Vol. 67 of *Communications in Computer and Information Science,* pp. 137–150. Springer. International Conference, ICAART 2009, Porto, Portugal, January 19-21, 2009.

[10] Simpson, R. M. (2007). Structural Domain Definition using GIPO IV. In *Proceedings of the Second International Competition on Knowledge Engineering.* Providence, Rhode Island, USA.

[11] Simpson, R. M., Mccluskey, T. L., Zhao, W., Aylett, R. S., & Doniat, C. (2001). An Integrated Graphical Tool to support Knowledge Engineering in AI Planning. In *Proceedings of the 6th European Conference on Planning.*

[12] Vaquero, T. S., Silva, J. R., Tonidandel, F., & Beck, J. C. (2011). itSIMPLE: Towards an Integrated Design System for Real Planning Applications. To appear in: *The Knowledge Engineering Review Journal,* special issue on International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS): Cambridge University Press.

[13] Vaquero, T. S., Romero, V., Tonidandel, F., & Silva, J. R. (2007). itSIMPLE2.0: An integrated Tool for Designing Planning Environments. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS 2007).* Providence, Rhode Island, USA.

[14] Vaquero, T. S., Silva, J. R., Ferreira, M., Tonidandel, F., & Beck, J. C. (2009). From Requirements and Analysis to PDDL in itSIMPLE3.0. In *Proceedings of the Third ICKEPS, ICAPS 2009,* Thessaloniki, Greece.

[15] Vaquero, T. S., Tonaco, R., Costa, G., Tonidandel, F., Silva, J. R., & Beck, J. C. (2012). itSIMPLE4.0: Enhancing the Modeling Experience of Planning Problems. In *Proceedings of ICAPS 2012 System Demonstration*. Atibaia, Sao Pualo, Brazil, pp. 11–14.

[16] Vodrazka, J., & Chrpa, L. (2010). Visual Design of Planning Domains. In *Proceedings of ICAPS 2010 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, pp. 68–69.

[17] Fratini, S., Pecora, F., & Cesta, A. (2008). Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. *Archives of Control Sciences,* 18(2), 231–271.

[18] Simpson, R. M., McCluskey, T. L., Long, D., & Fox, M. (2002). *Generic Types as Design Patterns for Planning Domain Specification.* In Proceedings of AIPS'02 Workshop on Knowledge Engineering Tools and Techniques for AI Planning, Toulouse, France.