Open World Planning as SCSP

Alexander Nareyek

GMD FIRST
German National Research Center for Information Technology
Research Institute for Computer Architecture and Software Technology
Kekuléstraße 7
D - 12489 Berlin, Germany

alex@ai-center.com

Abstract

Planning for the real world requires the ability to sense and reason about an arbitrary number of entities and relations that are not known in advance. However, most satisfaction-based planning systems only reason about a fixed number of given unique fluents. This article presents the planning model of the constraint-based EXCALIBUR agent's planning system. The model is based on structural constraint satisfaction, which makes it possible to use arbitrary plan structures and to handle open world planning.

Introduction

Conventional planning approaches use highly specific representations and algorithms, e.g., STRIPS (Fikes & Nilsson 1971), UCPOP (Penberthy & Weld 1992) and PRODIGY (Veloso et al. 1995). Newer approaches are often based on more general search frameworks like propositional satisfiability (SAT), operations research (OR) and constraint programming (CP). Examples include SATPLAN (Kautz & Selman 1996), ILP-PLAN (Kautz & Walser 1999) and CPlan (Van Beek & Chen 1999). The advantage of using a general framework instead of specific approaches is the availability of ready-to-use off-the-shelf methods and the more general applicability of the new methods developed for the specific domain. In addition, future changes in the problem specification must be reflected only at the modeling level and not in the underlying search algorithms.

However, the general search frameworks require a given structure of variables and constraints and search "only" for consistent values for the variables. For the task of planning, though, there are lots of possible structures. Thus, a kind of maximal structure is used in which unnecessary parts can be deactivated by certain value assignments. If a solution cannot be found, the structure is further expanded to allow for longer plans and the search starts again. But the (exponential) structure expansion is only feasible for small problems. For an open planning world, where it is not clear in advance how many and which types of fluents/resources are involved[1], the use of these maximal structures is a pointless approach.

The approach of structural constraint satisfaction (Nareyek 1999a; Nareyek 1999b) combines conventional constraint satisfaction with structural requirements and enables us to formulate and solve combinatorial search problems without explicitly giving the solution's structure. The approach is used in this paper to tackle planning problems without a closed-world assumption.

The planning problem is specified as a so-called *structural constraint satisfaction problem (SCSP).* From the SCSP, productions can be automatically derived that can be used to create/modify the structure of conventional constraint satisfaction problems (CSPs). In the search for constraint satisfaction, local search techniques are applied. The constraints are enhanced by domain knowledge and can vary their variables' values and apply productions to change the CSP's structure so as to improve their current satisfaction. Specific *structural constraints* must be maintained during this process.

(Nareyek 2000) presents an SCSP formulation for planning that is restricted to finite planning worlds. This article presents an extended version capable of handling open planning worlds.

The Model's Basics

This section introduces the model's basic concepts. A formal specification will be given in a later section.

The model focuses on resources. A resource (also called *state variable* or *fluent*)[2] is a temporal projection of a specific property's state, which may be subject to constraints such as preconditions and changes. Numerical as well as symbolic properties are uniformly treated as resources. For example, a battery's POWER and the state of a DOOR are resources:

[1]Incomplete knowledge can be related to various other features, like an uncertain outcome of actions. Here, however, we consider only issues related to the existence of entities. Other aspects of incomplete knowledge are handled by certain value assignments within the model (see (Nareyek 1998))

[2]The term *resource* is a common term in the CP/OR community and is used here because of the planning system's close connection with applications for resource allocation/optimization.

POWER is [0: $t \in [0..5]$, $10 - 0.75 \times t$: $t \in [6..13]$, 0: $t \in [14..\infty[$],

DOOR is [OPEN: $t \in [0..45]$, CLOSED: $t \in [46..60]$, UNKNOWN: $t \in [61..\infty[$].

Actions, Action Tasks and Action Resources

The execution of an **action** (like EAT PEANUT) includes **action task** subcomponents. These action tasks represent different operations that are necessary to carry out the actions. Each of the action tasks utilizes an **action resource** for its execution. For instance, the action EAT PEANUT requires action tasks on a MOUTH and a LEFT HAND or RIGHT HAND action resource. Figure 1 visualizes the assignment of action tasks to action resources.

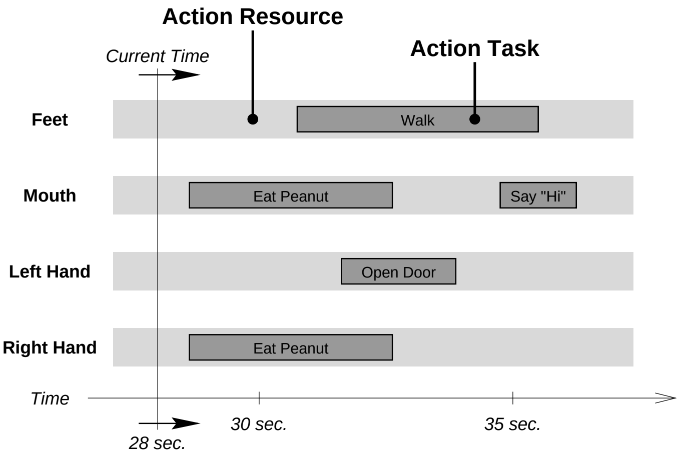

Figure 1: The Assignment of Action Tasks to Action Resources

It is forbidden for action tasks on the same action resource to overlap, as simultaneous executions of tasks would interfere with each other. For example, the agent is not allowed to talk and eat with his MOUTH at the same time.

The tasks of an action are subject to action-specific conditions. For example, the action tasks of the action EAT PEANUT must begin and end at the same time, and the begin and end values must be four seconds apart.

State Resources, State Tasks and Precondition Tasks

A **state resource** is similar to an action resource. It does not manage actively planned actions, but rather the development of a specific property of the environment or the agent itself. For example, an OWN PEANUT state resource with a Boolean assignment for any one time can provide information about the possession of a peanut (see Figure 2).

The status of the state resources can restrict the application of actions. To execute the action EAT

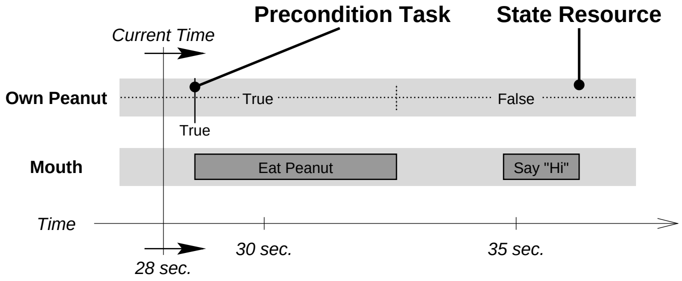

Figure 2: A State Resource

PEANUT, it is first necessary to have a peanut. These relations are checked by **precondition tasks** of actions. A precondition task includes a state value (or value ranges) that must correspond with the state of a related state resource at a specific time.

The effects of actions are more complicated to realize, as multiple actions and events may have synergetic effects. For example, a state resource HUNGER with assignments of natural numbers can be influenced by a beneficial action EAT PEANUT and a detrimental WALK at the same time.

It is the job of **state tasks** to describe an action's effects. For instance, a state task of the action EAT PEANUT is responsible for a decreasing **contribution** of -3 to the state resource HUNGER during the action's execution (see Figure 3). Each state resource has a specific **state mapping** that maps the contributions of the state tasks to values of the state resource's domain. In the case of the HUNGER resource, the synergetic effect is a simple addition of the state tasks' single gradients.

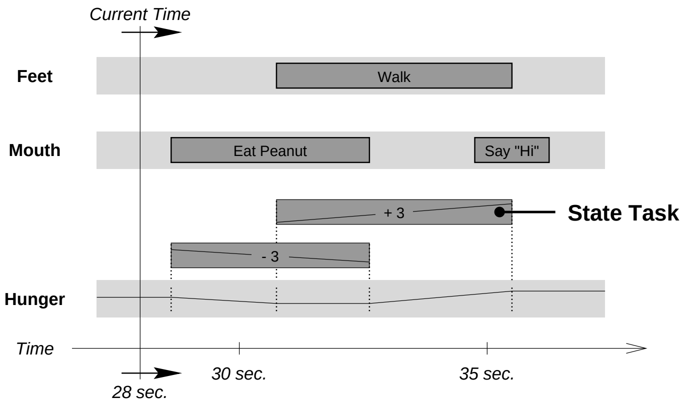

Figure 3: The Mapping Mechanism of State Resources

There can be further effects, which may be caused by synergetic effects within a state resource. Adding water to a bathtub may result in its overflowing and wetting the bathroom. The actions cannot provide state tasks to realize these further effects because an action has only the limited view of its state task contributions. Thus, **dependency effects** of specific state resource states must be expressed in addition. The dependencies are special actions that are beyond the agent's control. Expected external events can also be integrated by these dependencies.

Objects and References

In a finite and known world, there is a fixed set of resources to be considered. Static relations to specific state resources can be used to realize the effects of actions, e.g., that the state tasks of the action EAT PEANUT affect the state resources PEANUT LOCATION and PEANUT NUTRITIVE VALUE. But such static relations are no longer possible in an open world, where it is unclear which and how many resources exist[3]. Thus, as a consequence of the open world assumption, an action's tasks must be specified with variable **references** for the state resources involved (e.g., EAT X instead of EAT PEANUT).

The next problem that arises by dropping the closed-world assumption is that the relations between the state resources themselves are no longer fixed. For example, there could be two peanuts, a big and a small one, and thus multiple state resources PEANUT LOCATION and PEANUT NUTRITIVE VALUE. The resources of the same type are indistinguishable, and it is not clear which two belong to a specific peanut. If the EAT PEANUT action were applied, it is not clear which resources would be affected, and the big peanut might vanish, while the small one would be used to decrease hunger. Thus, in addition to an action's tasks, the state resources' states may involve references, too.

The most common relation between state resources is an aggregation — they form **objects**. As this is a very important relation, it is not handled by references, but explicitly represented in the model. For example, the state resources PEANUT LOCATION and PEANUT NUTRITIVE VALUE form an object PEANUT. Figure 4 illustrates the application of the EAT action to a PEANUT object.

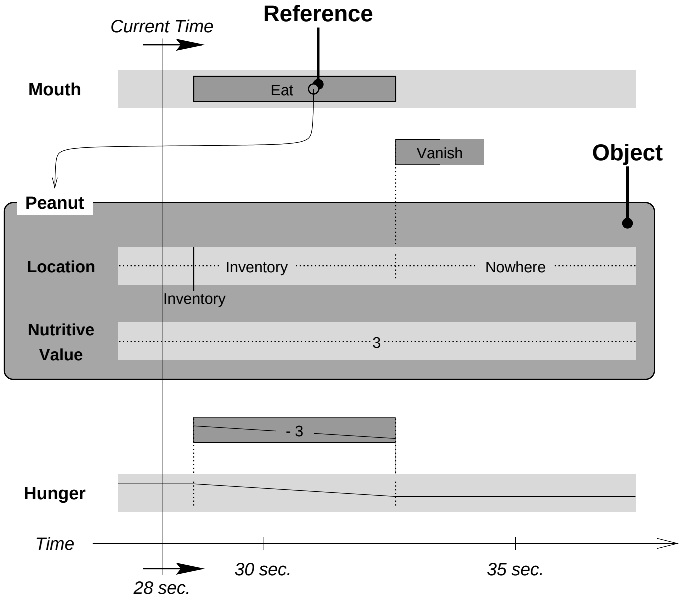

Figure 4: References and Objects

Sensors and Existence

The agent must be capable of acquiring new information about the environment, which must somehow be integrated into the planning process. The real-world data is collected by so-called **sensors**. The sensors report actual data, like the current level of hunger or the properties of a sighted peanut. We assume high-level sensoring that provides ready-structured objects. Sensors are related to the virtual objects of the plan. Figure 5 shows an example of a plan to put one block on top of another where only one of the blocks has already be sensed.

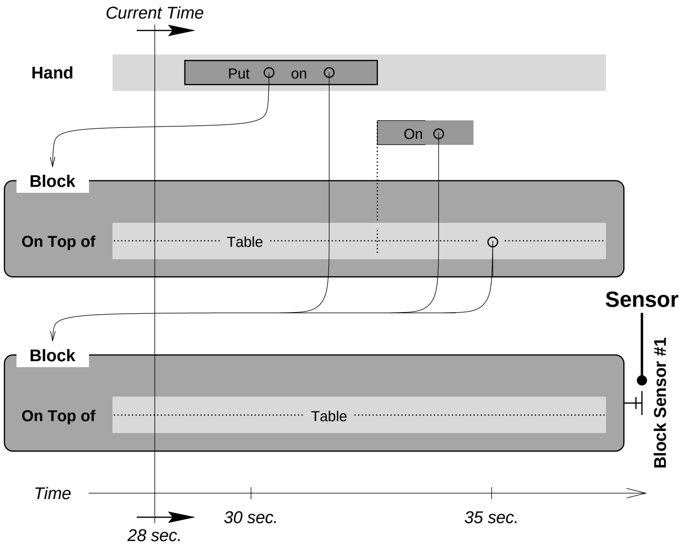

Figure 5: Sensors

Having introduced the concept of sensors, we are faced with the question of whether a planning object will ever be connected to a sensor, i.e., if a counterpart in the real world actually exists or will exist. For example, it is pointless optimistically creating/revising a plan so that a matching KEY is always found next to a locked DOOR. However, the existence of objects is not a yes/no matter. It is temporally dependent (a KEY may become available *after* a LOCK is installed) and a probabilistic matter (the KEY *may* become available). Thus, we need a temporally projected probabilistic measure for every object, which expresses the confidence that this object really exists — an **existence projection**.

[3] We assume that all possible types of resources are known, but not the number of instances.

Structural Constraint Satisfaction

This section describes the concept of structural constraint satisfaction. For a more detailed presentation, please refer to (Nareyek 1999a) and (Nareyek 1999b). The concept is based on algebraic graph grammars (Ehrig, Pfender & Schneider 1973; Rozenberg 1997).

A structural constraint satisfaction problem can be used to overcome the deficiencies of conventional CSPs with respect to structural variety. In an SCSP, the constraint graph is not explicitly given. Only the types of constraints (together with their possible connections) and structural constraints that restrict certain graph constellations are specified.

A CSP can be represented by a graph, in which the variables are circular nodes and the constraints rectangular nodes, an edge between a variable and a constraint indicating the variable's involvement in the constraint. Since a constraint may involve multiple variables, there must be a way to indicate a variable's role in the constraint. The direction — and a possible label — of an edge can be used for this purpose.

For structural constraint satisfaction, a distinction is made between *extensible constraints* and *nonextensible constraints*, as there may be constraint types that allow a variable number of elements to be included. Furthermore, an SCSP allows the existence of so-called *object constraints*. These do not restrict the variables' values, but merely provide structural context information. Object constraints are represented by a rectangular vertex with a dashed outline (see Figure 6).

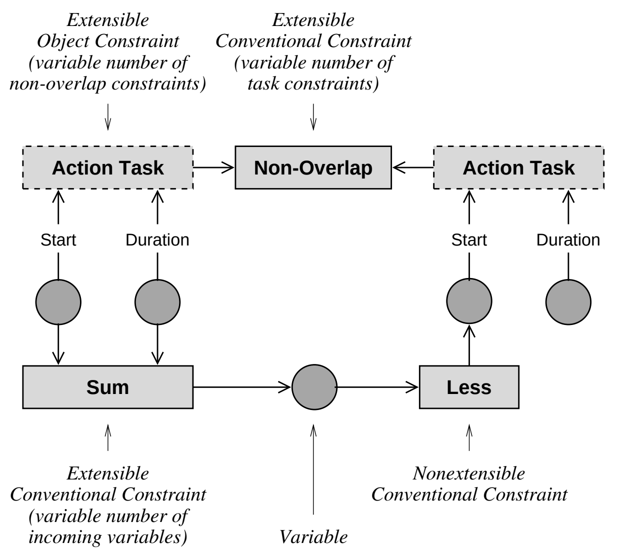

Figure 6: An Example Graph

Structural constraints allow us to formulate restrictions on admissible constraint graphs, e.g., that the same person is not allowed to drive a car and read a paper at the same time (see Figure 7). A structural constraint consists of a docking part and a set of testing-part alternatives. If the docking part of a structural constraint matches the constraint graph at some point, an alternative of the testing part must match, too. Otherwise, the graph is structurally inconsistent. An identity of objects in the docking part and the testing part is marked by identifiers like `:1`.

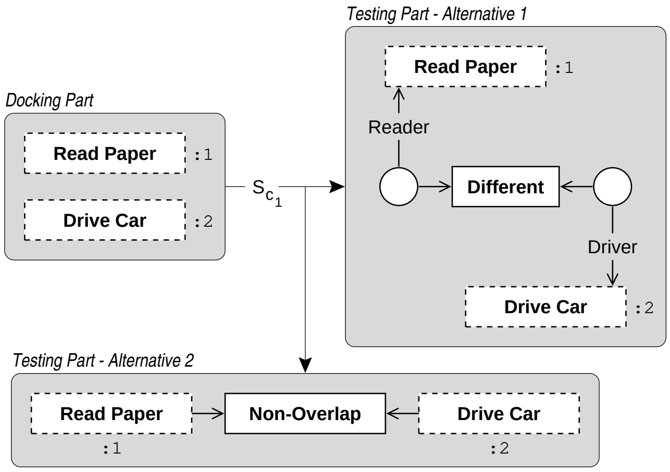

Figure 7: An Example of a Structural Constraint

Structural constraints may involve application conditions. A negative application condition (NAC) specifies a structure that is not allowed to match the graph. An NAC is indicated by a convex dark area (e.g., see Figure 11). For multiple application conditions, the conjunction of the conditions must hold.

A structural constraint satisfaction problem $SCSP = (\mathcal{CD}, \mathcal{S})$ consists of a tuple of sets of constraint descriptions $\mathcal{CD} = (\mathcal{C}_n, \mathcal{C}_e, \mathcal{O}_n, \mathcal{O}_e)$ and a set of structural constraints $\mathcal{S}$. The constraint descriptions of $\mathcal{C}_n$ and $\mathcal{O}_n$ are pairs (c, p_{base}) with a nonextensible conventional (or object) constraint c and its embedding graph p_{base}. The constraint descriptions of $\mathcal{C}_e$ and $\mathcal{O}_e$ are 4-tuples $(c, p_{base}, E, p_{max})$ with an extensible conventional (or object) constraint c, its minimal embedding graph p_{base}, a set of extension graphs E and the constraint's maximal embedding graph p_{max}.

An embedding graph shows the constraint with all its directly connected neighbor vertices. If an extensible constraint has no maximal embedding, p_{max} is the empty graph. An extension graph shows the constraint connected to the vertices that can be added in one step.

From an SCSP formulation, graph productions and further structural constraints to prevent redundancies can be automatically derived (Nareyek 1999a; Nareyek 1999b). The productions generate the structural search space (see Figure 8). However, certain parts of the search space are inconsistent because of the structural constraints.

The search mechanism used for the EXCALIBUR agent's planning system is based on local search. Conventional constraints have constraint-specific cost/goal functions which express the constraints' satisfaction/optimization. In addition, a constraint has in-

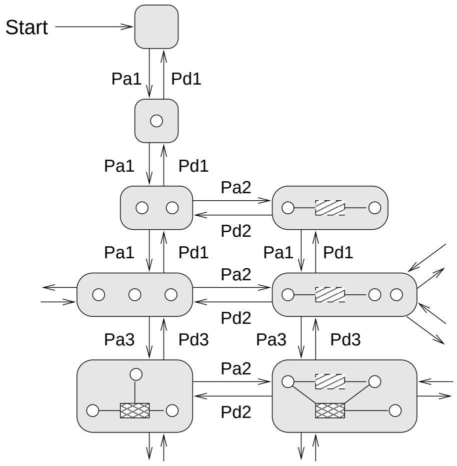

Figure 8: An Example of a Structural Search Space

ternal heuristics to improve its current state. These heuristics can perform knowledge-based changes in the variables' values and can also use productions for making structural changes in the graph. The constraint that is to improve the current state is selected by a so-called *global search control*.

In this way, the plan is improved step by step toward a consistent and optimal plan. Because of the application domain, computer games, the available reasoning time is very limited. Thus, the structural constraints are not checked during search, but only heuristics that do not threaten any structural constraints by structural changes may be used. This property of the heuristics must be ensured a priori.

The Planning Model as SCSP

This section defines the planning model in terms of structural constraint satisfaction. The representation of the planning model as an SCSP allows us to apply the techniques of (Nareyek 1999a; Nareyek 1999b) to generate the structural search space. Figure 9 gives an overview of possible relations between the planning SCSP's elements.

The Current Time

The very first thing we need is a variable for the current time because the constraints' heuristics and cost/goal functions use this as input. For example, actions that are still to be executed should not be placed in the past. The variable is marked by a Current Time object constraint (see Figure 10). As there can only be one current time, we need a structural constraint to prevent there being multiple variables representing the current time (see Figure 11).

Actions

An action consists of a set of different preconditions, operations and resulting state changes. These elements are represented by tasks, i.e., there are Precondition

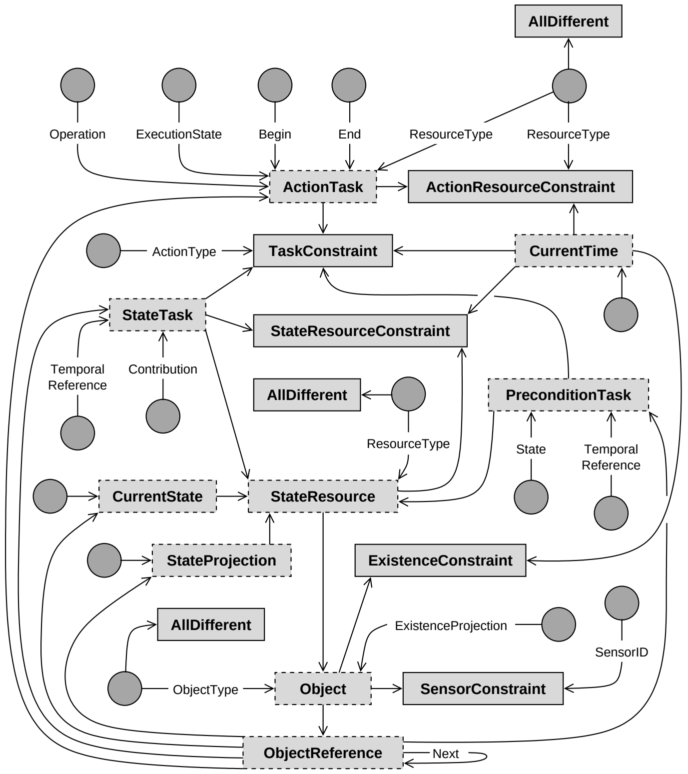

Figure 9: Possible Type Relations

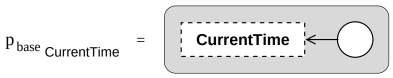

Figure 10: The Extensible Object Constraint Current Time

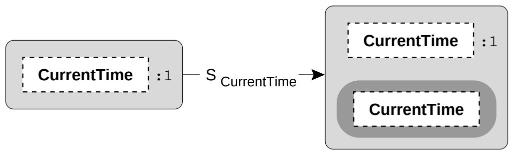

Figure 11: The Structural Constraint Current Time

Tasks for precondition tests, Action Tasks for operations and State Tasks for state changes. All tasks are represented by object constraints and must be connected to a Task Constraint.

A Task Constraint enforces a certain task configuration (i.e., it specifies which tasks are to be connected to the Task Constraint and what kind of restrictions apply to the tasks' variables) for a specific action, including the temporal order of the tasks. The specific action to be expressed is determined by the value of a connected *ActionType* variable (see Figure 12). The cost function of the Task Constraint describes the distance from the current task configuration to the configuration that is demanded by the *ActionType* variable. In addition, the *Begin* of nonexecuted Action Tasks before the Current Time is penalized.

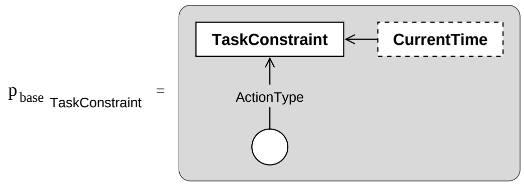

Figure 12: The Extensible Conventional Task Constraint

An action's task may not also be part of another action. This is ensured by the structural constraint in Figure 13 (an ellipse with a dotted outline matches any node).

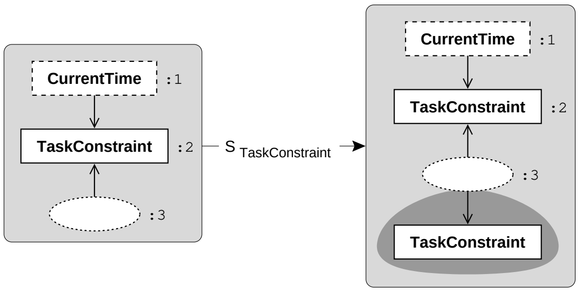

Figure 13: The Structural Task Constraint

Operations

An Action Task specifies a concrete operation that must be executed within an action (see Figure 14).

An Action Task's operation uses a specific action resource. For an Action Task's duration, other tasks are required to leave enough of the action resource's capacity to carry out the task's operation. An Action Resource Constraint (ARC) internally projects an action resource's capacity and reflects an overload of the resource by its cost function[4] (see Figure 15).

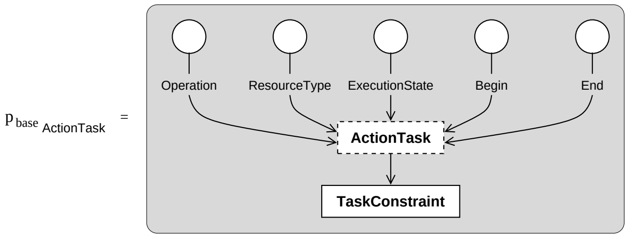

Figure 14: The Extensible Object Constraint Action Task

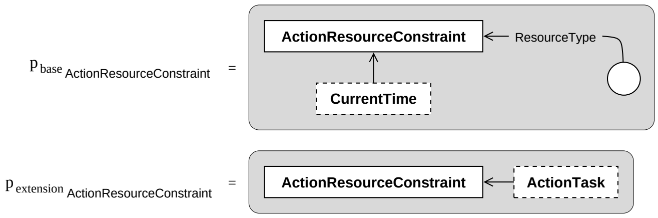

Figure 15: The Extensible Conventional Action Resource Constraint

Each Action Task must be linked to a specific ARC of the required *ResourceType* (see Figure 16). The Action Task that is to be executed at the Current Time on an action resource is determined by the ARC by demanding a special value for the task's *ExecutionState* variable.

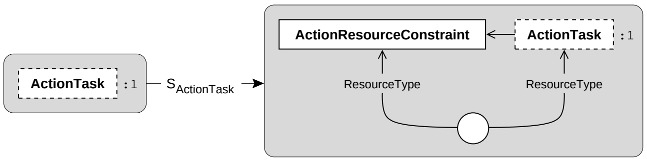

Figure 16: The Structural Constraint Action Task

In addition, all ARCs must have a different *ResourceType*. To realize this, we can use an All Different constraint (see Figure 17). This constraint ensures that all connected variables have different values. An ARC must have its *ResourceType* variable connected to an All Different constraint (see Figure 18). The requirement that there be no other All Different constraint for different ARCs is formulated later in the following sections.

The structural constraints also ensure that an Action Task cannot be connected to two ARCs at the same time.

States

Besides Action Tasks, an action consists of Precondition Tasks (see Figure 19) and State Tasks (see Figure 20). The reason why State Tasks do not have

[4]Multiple incorporation of the same Action Task is prevented by structural constraints that prohibit redundancies. These structural constraints are automatically deduced from the SCSP formulation (see (Nareyek 1999a)).

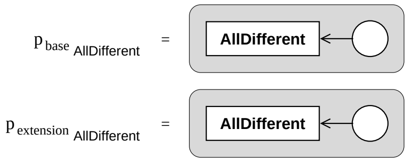

Figure 17: The Extensible Conventional Constraint All Different

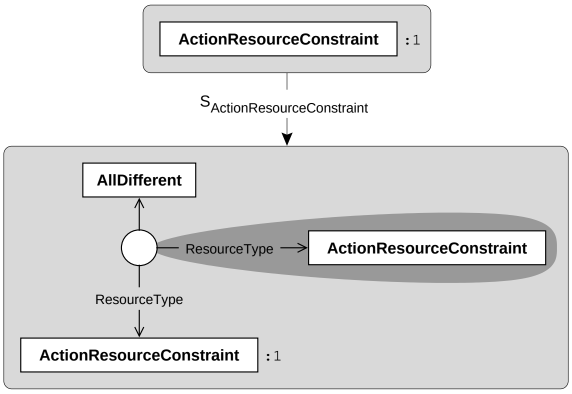

Figure 18: The Structural Action Resource Constraint

to be linked to a Task Constraint will become clear later.

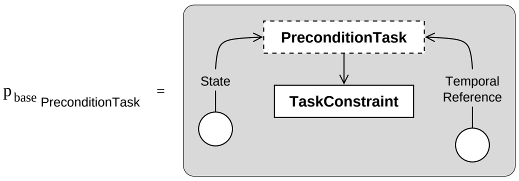

Figure 19: The Extensible Object Constraint Precondition Task

Unlike an action resource's structures, which are only internally represented in an ARC, a state resource's structures are explicitly stored in the model. This is because other constraints must access the state information as well. Thus, Precondition Tasks and State Tasks are linked to a State Resource object constraint which specifies the property that is to be tested/changed. A State Resource object constraint relates Precondition Tasks and State Tasks to a *ResourceType* variable, a State Projection and a Current State (see Figure 21).

All Precondition Tasks and State Tasks are required to be connected to exactly one State Resource (see Figures 22 and 23).

The Current State references a variable (see Figure 24) that contains the State Resource's state at the Current Time. The State Projection references a variable (see Figure 25) that stores the temporal projection of the resource's state for the whole timeline.

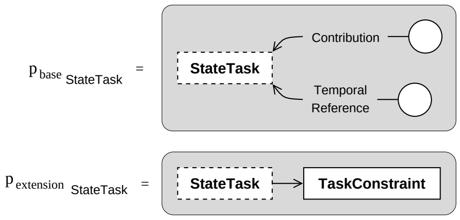

Figure 20: The Extensible Object Constraint State Task

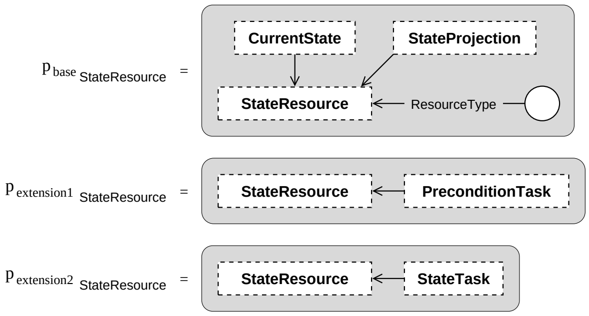

Figure 21: The Extensible Object Constraint State Resource

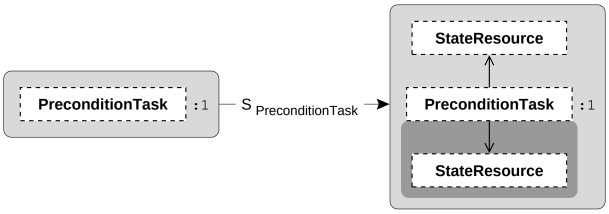

Figure 22: The Structural Constraint Precondition Task

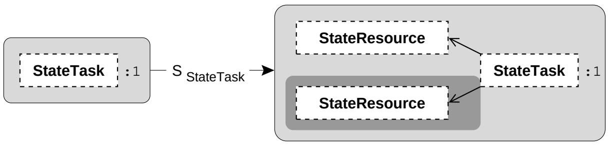

Figure 23: The Structural Constraint State Task

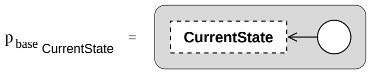

Figure 24: The Extensible Object Constraint Current State

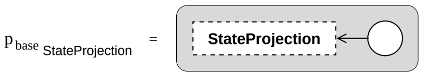

Figure 25: The Extensible Object Constraint State Projection

Both, the Current State and the State Projection, must be connected to exactly one State Resource (see Figures 26 and 27).

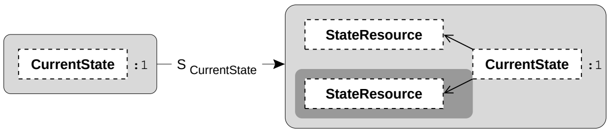

Figure 26: The Structural Constraint Current State

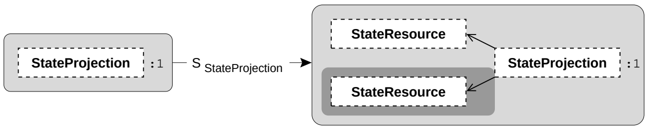

Figure 27: The Structural Constraint State Projection

A State Resource Constraint (SRC) is linked to the State Resource to ensure a correct State Projection (see Figure 28). The SRC uses the State Resource's Current State and the *Contributions* of the State Resource's State Tasks to project the property's state development according to the State Tasks' *TemporalReferences* on a timeline, which is stored in the State Projection's variable. The constraint's costs are computed according to satisfaction of the assigned precondition tests.

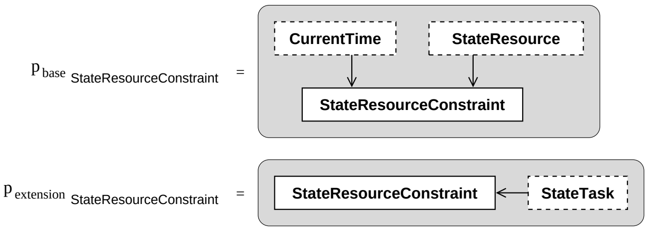

Figure 28: The Extensible Conventional State Resource Constraint

The structural constraint of Figure 29 ensures that each State Resource is connected to exactly one Object and that an SRC is connected. Although it is not necessary that all *ResourceType* variables of the State Resources have different values, this facilitates searching, e.g., for all Nutritive Value State Resources of the current plan. For this purpose, all *ResourceType* variables must be connected to an All Different constraint.

The SRC is also responsible for maintaining the dependency effects of the resource's state development. To accomplish this, State Tasks can be connected to an SRC. Thus, a State Task must either be connected to exactly one State Resource Constraint or one Task Constraint. This is ensured by the structural constraint of Figure 30 (together with that of Figure

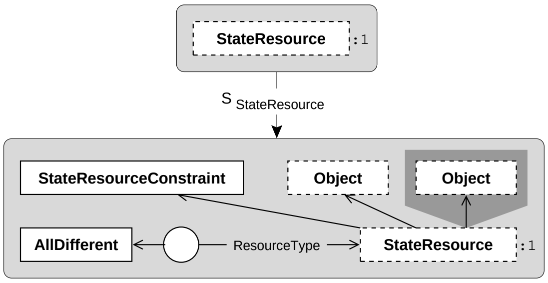

Figure 29: The Structural Constraint State Resource

13). A configuration of an SRC's State Tasks that does not represent the correct dependency effects has an impact on the value of the SRC's cost function.

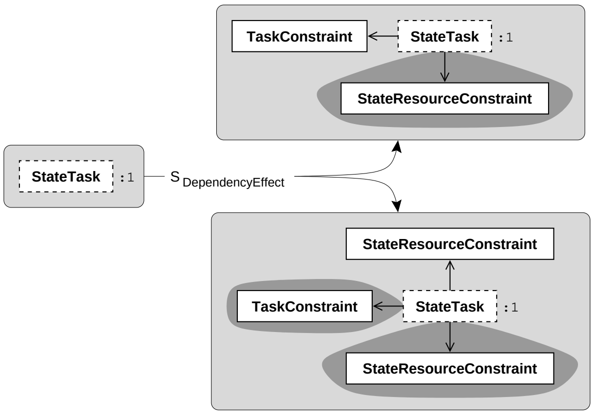

Figure 30: The Structural Constraint Dependency Effect

Objects

The Object aggregation is shown in Figure 31. The role of the *ExistenceProjection* variable and the Existence Constraint are explained in a later section.

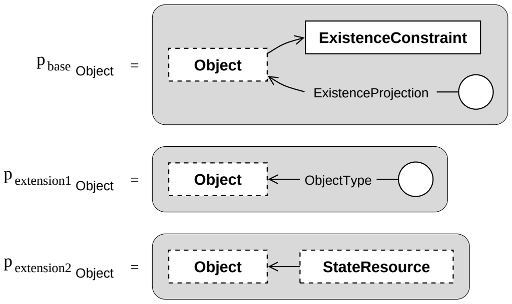

Figure 31: The Extensible Object Constraint Object

The *ObjectType* variable specifies the type of the Object — for example, a Door. The *ObjectType* variable is not directly included in the p_{base} graph, as this allows the search (using the production generation of (Nareyek 1999a; Nareyek 1999b)) to exchange an Object's *ObjectType* for a similar one without deleting the whole Object, e.g., if the search decides to consider a Picklock instead of a Key. However, this necessitates the structural constraint of Figure 32, which ensures that an Object has exactly one *ObjectType* variable. In addition — as with the SRC — all *ObjectType* variables are connected to an All Different constraint.

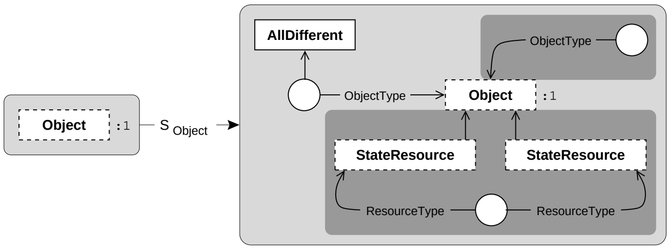

Figure 32: The Structural Constraint Object

Instead of a general unique identifier, the *ResourceType* of a State Resource has the task of specifying the State Resource's role within the Object, e.g., a Door's Location, Lock or Color. To prevent ambiguities, all *ResourceType* variables of an Object's State Resource must be different (see Figure 32).

This means, for an All Different constraint, that the constraint must either be connected to an ARC's or State Resource's *ResourceType* variable or an Object's *ObjectType* variable, and that there must be no other All Different constraint for the same kind of type variables (see Figure 33).

Any State Resource must be linked to an Object (see Figure 29). Agent-internal State Resources, e.g., the agent's Hunger, can be linked to a unique Ego Object.

A Task Constraint (or SRC in the case of dependency effects) must also ensure that the connection of *ResourceTypes* and *ObjectTypes* with the Task Constraint's (SRC's) State Tasks and Precondition Tasks is correct with respect to the Task Constraint's *ActionType* (*ResourceType* of the SRC's State Resource). For example, an action Eat X should link its State Task with a *Vanish* contribution to the same Object that its Precondition Task with the location test is linked to.

References

All tasks can use Object References to Objects, e.g., a general State Task that causes one Object X to be on top of another Object Y can make use of Object References to assign its variables X and Y. Likewise, the Current State and the State Projection can have Object References (see Figure 34). An Object Reference can also be linked to another Object Reference to realize a list. For example, the State Task of the previous example would

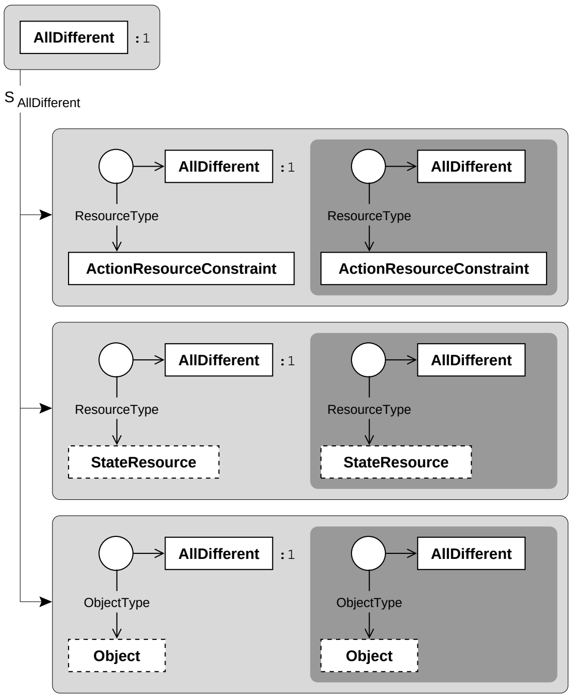

Figure 33: The Structural Constraint All Different

have to use a list (instead of a set) with two Object References to distinguish the two Objects (which Object has to be placed onto which). The Task Constraint (or SRC in the case of dependency effects) must ensure that the reference structures are valid.

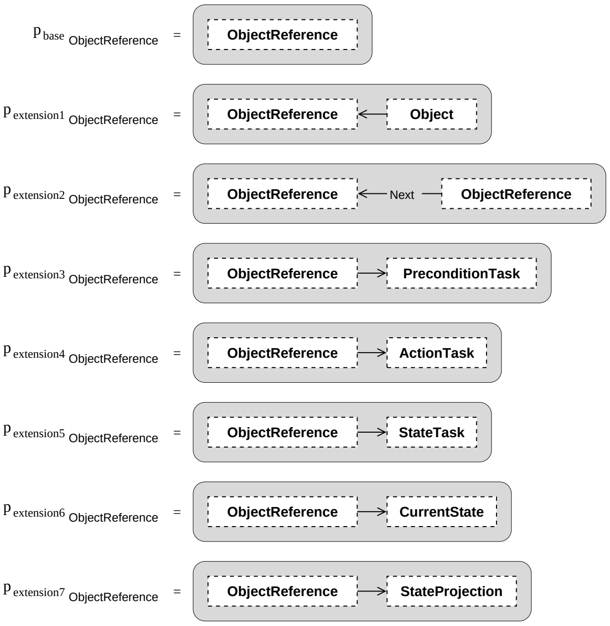

Figure 34: The Extensible Object Constraint Object Reference

All Object References are required to be connected to exactly one Object that is to be referenced (see Figure 35).

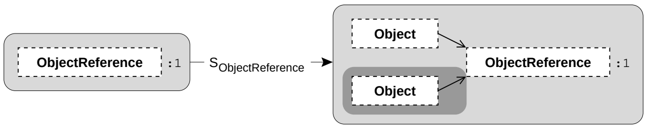

Figure 35: The Structural Constraint Object Reference

Sensors

The sensors provide data that is structured according to the planning model's Objects. The linking of a sensor with an appropriate Object is done by a Sensor Constraint that links a *SensorID* variable, which specifies the real-world's sensor, with an Object (see Figure 36). The Sensor Constraint ensures that the Object's *ObjectType* variable corresponds to the sensor data and that the Object has all necessary State Resources. The productions for an addition/deletion of a Sensor Constraint are not allowed to be applied by any improvement heuristic. If the Sensor Constraint is satisfied, the sensor data can provide values for the *CurrentState* variables of the connected Objects' SRCs.

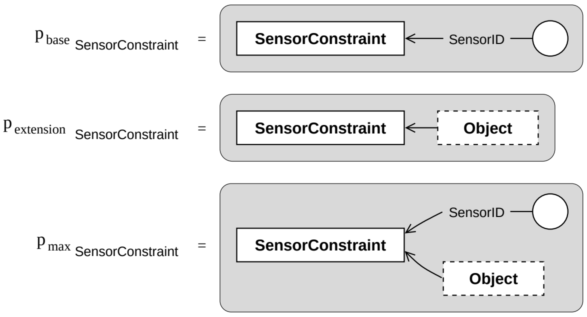

Figure 36: The Extensible Conventional Sensor Constraint

Again — as with an Object's *ObjectType* — a Sensor Constraint's Object is not directly included in the p_{base} graph, as this allows the search to exchange a Sensor Constraint's Object for a similar one without deleting the whole Sensor Constraint, e.g., if the search decides that a sensed Peanut is a new one instead of the Peanut that has been eaten. However, this necessitates the structural constraint of Figure 37, which ensures that a sensor is linked to an Object.

Existence Projections

The confidence projection that an Object really exists is realized by using an *ExistenceProjection* variable, which is similar to the temporal *StateProjection* variables of SRCs, taking values between 0 and 1 at each time point (see Figure 31).

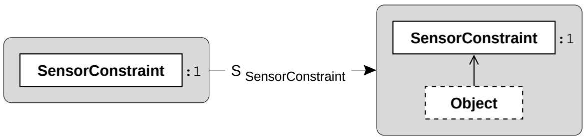

Figure 37: The Structural Sensor Constraint

The *ExistenceProjection* is not only dependent on Object-local features but on the whole world state. For example, it is improbable that there is a Door if the agent's Location is Park. Given these global conditions, an Existence Constraint must be linked to all Objects to be capable of accessing all relevant State Resources. To keep the linking costs reasonably low, only one Existence Constraint is responsible for maintaining all Objects' *ExistenceProjections*.

Figure 38: The Extensible Conventional Existence Constraint

Figure 39: The Structural Existence Constraint

The Existence Constraint must also ensure correct Object configurations. For example, an Object with a Door *ObjectType* and a Nutritive Value State Resource gets a very poor *ExistenceProjection*. However, Objects do not have to be specified completely. For example, the Door's Color State Resource may not be important for a plan to open it.

The *ExistenceProjection* of an Object has an impact on the satisfaction of Task Constraints that have tasks connected to the Object.

Problem Formulation

The SCSP of the previous sections specified a general planning problem. For a specific problem, the constraints must be able to handle the specific domain values and must have appropriate cost/goal functions and improvement heuristics, e.g., the Task Constraint must know about the permitted action configurations and the Existence Constraint must be able to project the existence confidence for the domain's Objects.

A specific planning problem can include satisfaction goals (e.g., the door *is to be open* at time point 2507) and optimization goals (e.g., the maximal hunger level over time *is to be as low as possible*). Satisfaction goals can be represented by TASK CONSTRAINTS that consist only of PRECONDITION TASKS, and optimization goals can be realized by initialization of the resource constraints' goal functions[5].

The satisfaction goals and optimization goals must be embedded in the search process. Thus, we can define additional constraints that restrict variables to a specific value (a constant) and require that these constraints exist in a solution where they, for example, restrict the TASK CONSTRAINTS' *ActionType* variables and STATE RESOURCES' *ResourceType* variables (e.g., see Figure 40).

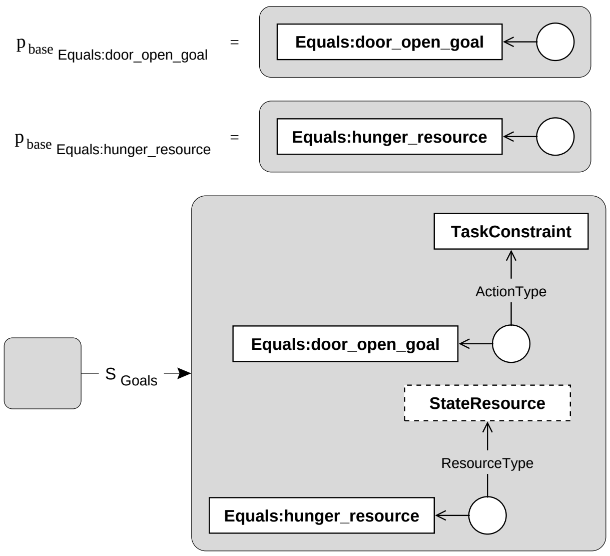

Figure 40: Specification of Satisfaction Goals

Conclusion

This paper presented the planning model for the EXCALIBUR agents. It combines temporal planning with resource reasoning and makes it possible to situate the agents in an environment where they have only a restricted view of the world state.

The planning model is based on the SCSP approach and features a domain-independent representation. In contrast, constraint-based planning systems like *parc*PLAN (El-Kholy & Richards 1996) and CPlan (Van Beek & Chen 1999), besides their inability to search in a way that is not focused on the criterion of plan length, do not have a model for general planning, but rely instead on domain-dependent encodings.

The past decade has seen the development of a few planning systems capable of reasoning about incomplete world states, e.g., XII (Golden, Etzioni & Weld 1994) and PSIPLAN (Babaian & Schmolze 1999). Satisfaction-based planners in this domain are very rare. The lack of satisfaction-based approaches to open world planning can be explained by the massive explosion of the search space under consideration. Satisfaction-based planners usually work on maximal structures, which become unmanageable for more complex problems. The presented planning model avoids the use of maximal structures by including the search for the structure as part of the satisfaction process.

The planning model borrows from typical constraint-based applications for resource allocation/optimization. The power of global constraints for constraint-specific representation and reasoning mechanisms for specific resource types was recognized here very early on and led to significant speed-ups in the solution process. General frameworks for planning and scheduling like Muscettola's HSTS (Muscettola 1994) lack such specialized representation and reasoning capabilities.

More information on the underlying EXCALIBUR project is available at:

`http://www.ai-center.com/projects/excalibur/`

[5]Of course, the whole expressiveness of the SCSP approach can be used to formulate much more complicated goals, but for most cases, a representation using specific TASK CONSTRAINTS / resources' goal functions is adequate to model the goals.

Acknowledgments

The work reported here is supported by the German Research Foundation (DFG), NICOSIO, Conitec Datensysteme GmbH and Cross Platform Research Germany (CPR).

References

Babaian, T., and Schmolze, J. G. 1999. PSIPLAN: Planning with ψ-forms over Partially Closed Worlds. In Proceedings of the Fifth European Conference on Planning (ECP'99).

Ehrig, H.; Pfender, M.; and Schneider, H. J. 1973. Graph Grammars: An Algebraic Approach. In Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT), 167–180.

El-Kholy, A., and Richards, B. 1996. Temporal and Resource Reasoning in Planning: the *parc*PLAN approach. In Proceedings of the Twelfth European Conference on Artificial Intelligence (ECAI-96), 614–618.

Fikes, R. E., and Nilsson, N. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 5(2): 189–208.

Golden, K., Etzioni, O., and Weld, D. 1994. Omnipotence Without Omniscience: Efficient Sensor Management for Planning. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), 1048–1054.

Kautz, H., and Selman, B. 1996. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), 1194–1201.

Kautz, H., and Walser, J. P. 1999. State-space Planning by Integer Optimization. In Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99), 526–533.

Muscettola, N. 1994. HSTS: Integrating Planning and Scheduling. In Zweben, M., and Fox, M. S. (eds.), *Intelligent Scheduling*, Morgan Kaufmann, 169–212.

Nareyek, A. 1998. A Planning Model for Agents in Dynamic and Uncertain Real-Time Environments. In Proceedings of the Workshop on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments at the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS'98), Technical Report, WS-98-02, 7–14. AAAI Press, Menlo Park, California.

Nareyek, A. 1999a. Structural Constraint Satisfaction. In Papers from the AAAI-99 Workshop on Configuration, Technical Report, WS-99-05, 76–82. AAAI Press, Menlo Park, California.

Nareyek, A. 1999b. Applying Local Search to Structural Constraint Satisfaction. In Proceedings of the IJCAI-99 Workshop on Intelligent Workflow and Process Management: The New Frontier for AI in Business.

Nareyek, A. 2000. AI Planning in a Constraint Programming Framework. In Hommel, G. (ed.), *Communication-Based Systems*, Kluwer Academic Publishers, 163–178.

Penberthy, J. S., and Weld, D. S. 1992. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92), 102–114.

Rozenberg, G. ed. 1997. *The Handbook of Graph Grammars. Volume I: Foundations.* Reading, World Scientific.

Van Beek, P., and Chen, X. 1999. CPlan: A Constraint Programming Approach to Planning. In Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99), 585–590.

Veloso, M.; Carbonell, J.; Pérez, A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating Planning and Learning: The PRODIGY Architecture. *Journal of Experimental and Theoretical Artificial Intelligence* 7(1).