SquadSmart
Hierarchical Planning and Coordinated Plan Execution for Squads of Characters

Peter Gorniak and **Ian Davis**
Mad Doc Software
100 Burtt Road, Suite 220
Andover, MA, 01810
{pgorniak,idavis}@maddocsoftware.com

Abstract

This paper presents an application of Hierarchical Task Network (HTN) planning to a squad-based military simulation. The hierarchical planner produces collaborative plans for the whole squad in real time, generating the type of highly coordinated behaviours typical for armed combat situations involving trained professionals. Here, we detail the extensions to HTN planning necessary to provide real-time planning and subsequent collaborative plan execution. To make full hierarchical planning feasible in a game context we employ a planner compilation technique that saves memory allocations and speeds up symbol access. Additionally, our planner can be paused and resumed, making it possible to impose a hard limit on its computation time during any single frame. For collaborative plan execution we describe several synchronization extensions to the HTN framework, allowing agents to participate in several plans at once and to act in parallel or in sequence during single plans. Overall, we demonstrate that HTN planning can be used as an expressive and powerful real-time planning framework for tightly coupled groups of in-game characters.

Introduction

Most non-player characters in today's computer games utilize finite-state or rule based designs to react to gameplay events. Major game engines even provide scripting languages that incorporate a finite state machine directly into the source code (Epic Games 2006). However, while it is possible to produce sophisticated multi-step intelligent behaviours with these methods, doing so becomes tedious and error-prone and does not generalize well across characters or to new situations. Recently, real-time planning has been suggested as an alternative to produce automatically generated multi-step behaviors for characters.

Beyond individual characters, however, groups of AIs often need to co-ordinate and work together. Neither state machines nor planners for individuals address this issue, and existing games employ largely ad-hoc solutions for group-based AI.

In this paper, we present SquadSmart, the AI controller for squads of characters used in an upcoming military training simulation. SquadSmart employs a Hierarchical Task Network (HTN) Planner to simultaneously plan for up to 4 members of a police unit. The HTN framework allows us to assign tasks to available squad members, decomposing them and selecting amongst multiple possible strategies automatically. Furthermore, the controller extends the HTN paradigm by specifying synchronization and plan execution components, thus allowing squad mates to wait for others to complete actions or to execute actions simultaneously while working on a shared plan. The planner bases its decisions on a common squad-level knowledge base that includes both individual knowledge (for example, which squad member can see which enemy) as well as global squad knowledge (for example, which squad member is assigned to which subsquad). Our system also includes a compiler that translates an HTN domain specification into C++ code, allowing optimizations for speed and memory use. We also introduce a method to distributed a single planning run over multiple frames in a game, providing the ability to cap planning time to a fixed fraction of the time spent to produce a frame.

This work demonstrates that hierarchical real-time planning for teams of artificial characters is feasible in a game context, and provides a succinct global behavior specification that engineers and designers can use to automatically generate complex coordinated behaviors for non-player characters.

Related Work

While the idea of planning has existed since the beginnings of the field of Artificial Intelligence, planning algorithms have only recently been applied to commercial interactive games: Orkin (2005) employed a STRIPS planner (Fikes, Hart, & Nilsson 1972) to give intelligence to enemies in the first person shooter F.E.A.R.. In the academic realm, HTN planning has been applied to pick strategies for Unreal Tournament bots (Hoang, Lee-Urban, & Munoz-Avila 2005) and Dini *et al.*(2005) provide an overview of various planning algorithms and how the might apply to computer games. Gorniak & Roy has previously employed plan recognition to provide contextual speech understanding in collaborative computer games (Gorniak & Roy 2005; in press)

For the game presented here, we use an HTN planning system (Erol, Hendler, & Nau 1994) based on SHOP2 (Nau *et al.* 2003), borrowing elements of the JSHOP2 planner compilation techniques (Ilghami & Nau 2003). The broader context of collaborative plans of which those discussed here are a subtype is discussed by Grosz & Kraus 1996.

Hierarchical Planning for Squads

We follow the SHOP (Simple Hierarchical Ordered Planner) in our domain specification and implementation, which means our planner plans for tasks in the same order that they will be accomplished. Similar to the SHOP planner, our planner supports various symbolic and numeric computations as well as external function calls.

HTN Planning

A *task* in HTN planning can be either compound or primitive in nature. *Compound tasks* decompose into further tasks, whereas *primitive tasks*, the leaves of the task network, can be directly executed. The *planning domain* consists of *methods*, which specify how to decompose compound tasks, *operators* which accomplish primitive tasks and *axioms*, which prove logical primitives without accomplishing tasks. Each method can have several branches, and there can be several operators for the same task, providing multiple ways of accomplishing a task. Each method branch and each operator has a precondition dictating whether it applies to the current state of the world or not. Only operators actually change the state of the world by adding to or deleting from it. The final plan is a linear sequence of such operators that accomplishes a given task, if such a sequence exists. The planner performs a search by decomposing tasks using the domain's methods and operators, gathering valid bindings for logical expressions as it proceeds.

As one of our goals in the design of SquadSmart was to provide a centralized and general planning mechanism and domain, we chose to use SHOP's general logical expression language instead of incorporating the logic of planning into game-specific C++ code. While this does mean that time and memory is spent on general logical proofs and the accompanying bookkeeping, it allows us to cleanly separate the domain specification and the planner from the game code itself, and to make SquadSmart re-useable for other game types. We address the performance issues via domain compilation and interruptible planning as discussed below.

The state of the world is specified as a collection of logical predicates denoted as $(p\ t_1\, t_2\, ...\, t_n)$ where p is the head of the predicate and the $t_1\, ...\, t_n$ are constants or numbers. Operators and methods, marked with the `:operator` and `:method` tags respectively, each contain a head, which is a predicate as above which can also contain variables as terms (denoted with a question mark as in $?x$). An operator also specifies a precondition (a logical expression which is evaluated using the current variable bindings and the current state) as well as an add and a delete list of ground predicates that will be added to or deleted from the world state when the operator is applied. A method consists of method branches, each branch in turn constituted by a precondition as well as a list of tasks into which this branch decomposes the parent task. We provide examples of planning domains in the next section.

HTN Planning Applied to Squads

During the game, a sensory system deposits knowledge into the HTN state representation. For example, squad members see enemies and doors, and can tell whether enemies are surrendered, restrained or dead. By default, the game runs the planner at fixed time intervals. Squad members can be part of any number of plans simultaneously, the only restriction being that they cannot perform two plan steps at the same time. Certain events can also force an immediate re-planning, such as a squad member dying. The planner's domain has a single top-level task with an associated method that prioritizes all possible sub-tasks. The planner will always select the first accomplishable method branch, and, adopting the SHOP semantics, it will in fact fail to find a plan if a method branch's precondition is fulfilled but the branch itself fails.)

Figure 1 shows the specification for the HTN methods accomplishing the restraining of surrendered suspects by squad team members. The general method `squad_restrain` finds the closest squad member to a restrained target. The helper method first attempts to find a second squad member close by that can perform a complement to the restrain (covering the event with raised weapon), or has the closest squad member perform the restrain alone if no suitable partner can be found. Most of the predicates directly match knowledge deposited in the state by the sensory system. Some are proven via axioms, for example `different`. The `call` term calls an external or a library function, here `MDActorDistance` to measure the distance between the squad member and the suspect to be restrain, and $<$ to decide whether the squad member is close enough to the suspect to perform an autonomous restrain. The `assign` term assigned the result of an expression to a variable (here `?d` for distance) where the `:sort-by` clause enforces the bindings for the following precondition to be sorted by the given variable's value, here again distance.

In this simple example, the helper method directly expands into primitive tasks, accomplished by operators. As in SHOP, we distinguish between internal (marked by "!!") and external (marked by "!") operators. Internal operators always have their effects applied and do not engage squad members in action, whereas external operators cause the squad member to perform an action on the game world, which may result in the state of knowledge changing. For example, restraining a suspect will cause the sensory system to add a `restrained` fact about that suspect to the world state. The `!!reserve` operator checks that its argument is not already marked as `reserved` and then marks it as reserved. This prevents a subsequent planning run from reserving the same target for a different purpose (or for the same purpose repeatedly.)

Figure 2 shows the domain graph for the squad planner. Decomposition of methods into other methods and operators is shown via arrows.

Planning Efficiency and Time-Slicing

Planning in general and HTN planning in particular is usually implemented as a recursive search process generating

```
(:method (squad_restrain_helper ?squad ?target)
  ;;-- PRECONDITION
  ;;restrain with complement
  ;;use second closest squad member to target
  (:sort-by ?d ((squadmember ?squad2)
                ;;make sure it’s a different squad member than the one
                ;;performing the restrain
                (different ?squad ?squad2)
                ;;make sure the squad member isn’t under other orders
                (not (following_order ?squad2))
                ;;measure the distance to the retrain target
                (assign ?d (call MDActorDistance ?squad2 ?target))
                ;;don’t perform a complement if too far away
                (call < ?d 400)))
  ;;-- TASK DECOMPOSITION
  (;;reserve all parties involved
   (!!reserve ?squad) (!!reserve ?squad2) (!!reserve ?target)
   ;;perform the restrain
   (!restrain ?squad ?target)
   ;;in parallel, perform the complement
   (!restrain_complement ?squad2 ?target)
   ;;at this point, do not go further until restrain is done
   (!!global_block ?squad)
   ;;free all parties involved
   (!!free ?squad) (!!free ?squad2) (!!free ?target))

  -- SECOND BRANCH (if first one’s precondition fails)
  ;;lonely restrain
  -- PRECONDITION
  () ;;all precondition elements for this case are covered in the parent
  -- TASK DECOMPOSITIOIN
  (;;reserve all parties involved
   (!!reserve ?squad) (!!reserve ?target)
   ;;perform the restrain
   (!restrain ?squad ?target)
   ;;wait till the restrain is done (so we don’t free the target prematurely)
   (!!global_block ?squad)
   ;;free everyone
   (!!free ?squad) (!!free ?target)))

(:method (squad_restrain)
  ;;-- PRECONDITION
  ;;use closest squad member to target
  (:sort-by ?d (;;find a surrendered target
                (surrendered ?target)
                ;;that’s also an enemy
                (enemy ?target)
                ;;pick a squad member
                (squadmember ?squad)
                ;;make sure the target is alive
                (not (dead ?target))
                ;;measure distance between squad member and target
                (assign ?d (call MDActorDistance ?squad ?target))
                ;;do not autonomously restrain if too far away
                (call < ?d 400)))
  ;;-- TASK DECOMPOSITION
  ;;we now know this is a valid restrain situation
  ;;the helper will decide whether to perform a complement or not
  ((squad_restrain_helper ?squad ?target)))
```

Figure 1: SHOP code of HTN methods for autonomous restraining

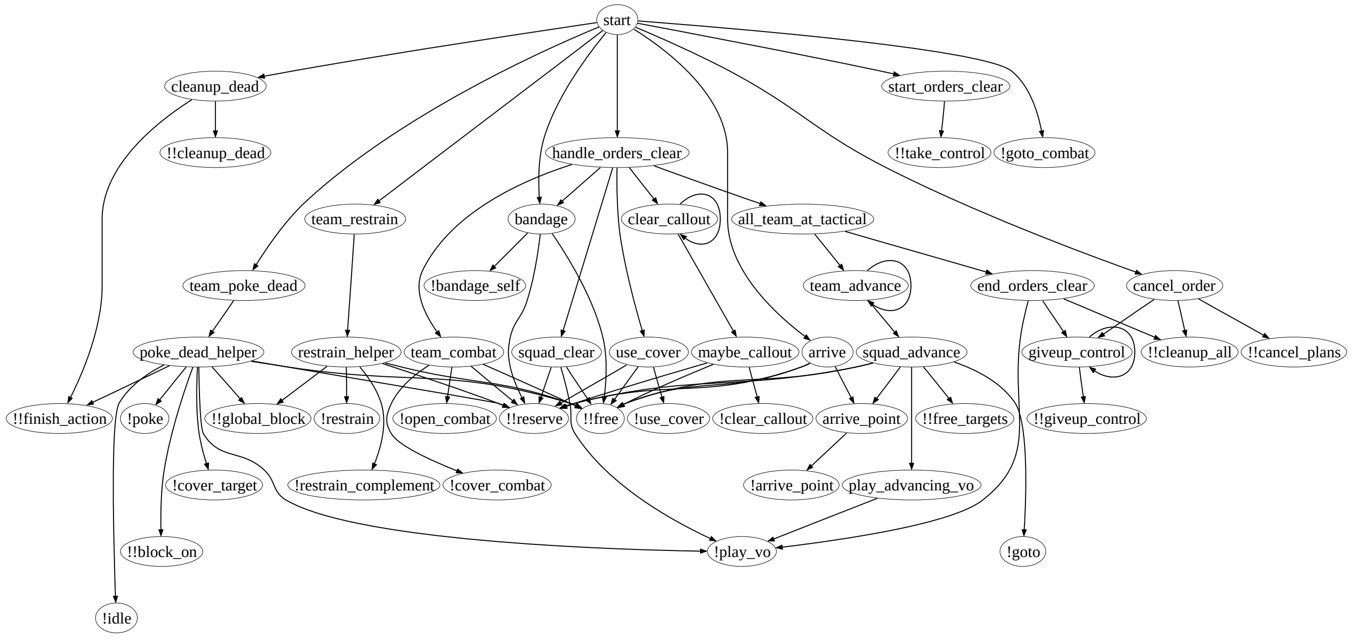

Figure 2: Relationships between Domain Methods, External and Internal Operators for the Squad Domain

possible plans. As such, it is hard to predict or limit how long any given planning search will take, especially in a setting where the world changes as dynamically as in a real-time game. One way to make planners more efficient is to make them highly specific to the planning problem in question. Orkin, for example, limits the planner in F.E.A.R. to only consider a small static array of typed variables that specify a world state only applicable to this particular game. However, doing so limits the generalizability of the planner - it needs to be heavily revised to function in another game and actual game data structures have to change if the scope of the world state changes during development. Furthermore, the knowledge and rules of the planner are embedded in the game's C++ code, making it harder to keep them modular and general.

A full logic representation for both planning domain and world state, on the other hand, implies a level of abstraction and indirection that sacrifices some efficiency in favour of gaining expressive power, generality and a clean separation from other game code. It also allows us to enforce a centralized knowledge representation that the planner works from, ensuring that when any given decision is made in the game we have easy access to the full state of knowledge that the decision was based upon.

To eliminate the problem of efficiency introduced by our decision, we adopt two strategies: domain compilation and time slicing of the planning process. Domain compilation is part of a planner compilation technique recently introduced for the JSHOP planner (Ilghami & Nau 2003), which compiles both the domain and the problem into a unique instance of the planner that solves exactly the one problem specified. Our problem and state are highly dynamic making it impossible to compile them into C++ code beforehand, but the domain is static and can be optimized during compilation. Introducing domain compilation lets us turn dynamic arrays into static, fixed-size arrays, allows us to replace arbitrary names for variables and predicates with C++ enumerations and makes it possible to pre-allocate most structures needed at the beginning of the game to avoid memory fragmentation and allocation overhead during planning. Note that the compilation technique is independent of the game itself, and can thus be re-used for other domains and games.

Domain compilation goes a long way towards making any single planning step more efficient, but it does not prevent deeps recursion during the generation of complex plans. If such a long planning session is performed during a single frame it produces a computation time spike for that frame, which is unacceptable in a game. To eliminate this problem, we have implemented time-slicing for our HTN planner, which allows us to pause the planning process at any given time, and resume it later. In this way, planning can be spread out over multiple frames, and we can guarantee that the planner never take more than its allocated time per frame, while still finishing planning in few enough frames to make characters responsible to game events. To be able to pause planning at any given time, we produce all logical bindings through generators that produce one binding at a time, eliminating recursion to produce nested bindings. In addition, our outer planning loop that drives the search process by recursively looking for the appropriate methods and operators to solve the current problem saves its stack. Each stack frame consists of

1. the current method or operator being tried
2. in the case of a method, the method branch being tried
3. the generator for the current operator or method branch's

precondition

4. the current list of tasks being solved
5. a flag indicating whether a satisfier has been found for the current method branch (no further branches should be tried from this method if this flag is true).

Once the planner has exceeded its allocated time during a given frame it begins unwinding its current recursion. On the next frame, it continues planning by re-using the saved stack frames at each level of recursion, arriving back at the place it left the search.

Plan Execution

Standard HTN planning specifies little about plan execution beyond the notion that primitive tasks can be carried out by some agent in the sequence specified by the plan. SHOP adds the distinction between internal and external operators, but stops there. In the scenario discussed here, however, real-time execution of plans involving several characters requires both the specification of execution semantics such as blocking on actions and synchronization as well as the handling of action failures and character deaths.

Execution Semantics

In-game actions may take some time. Executing an action can thus mark the squad member as busy. We assume that squad members can be busy with exactly one action at any given time. However, some actions may not mark squad members as busy despite taking time to execute. For example, voice overs may play while other actions are executed. Whether to mark a squad member as busy can also be parameterized - voice overs in some plans should play during other actions, while other may require waiting for voice over completion (e.g. squad member should only start advancing once the "area clear" has completed.)

When a squad member is marked as busy, all further actions involving this squad member are blocked, including actions that would not mark the squad member as busy. In the `restrain` method above, for example, the `!!free` operator on `?squad` will only execute once the squad member bound to the variable has finished its `!restrain` action. Actions involving other actors, however, can freely execute, such that in the example above the second squad member performs its `!restrain_complement` while the first is performing its `!restrain`. The planner continuously produces plans during plan execution so that squad members can be involved in several plans at any given time.

If an action does not mark a squad member as busy, it is executed and plan execution proceeds to the next action in the plan. The execution loop guarantees that all such actions are executed immediately, meaning before the world state changes or other actions finish or new plans are produced. Thus, if a plan is found consisting only of actions that do not engage actors, it executed completely in the same frame, not interleaving with any other plans. Guaranteeing immediate execution for these actions provides atomicity for operator groups such as the double *!!reserve* in the restraining example, making sure that it is not possible to have another plan reserve the target when the current plan has reserved the squad member.

Synchronization Facilities

According to the plan execution semantics covered so far, actions by different actors are always carried out in parallel, and general plan execution always proceeds despite a given character carrying out an action. Interpreting plans in this way allows the parallelism necessary for team action, but does not provide a way for one actor to wait for another, or for plan execution to halt until an actor finishes its action. To support such synchronization, we introduce several special internal operators:

`!!block_on` Takes two actors as arguments, and marks the first as busy until the second finishes the current action.

`!!global_block` Waits with any further execution of the plan it appears in until the actor specified as its argument finishes its current action. Other plans continue executing.

`!!finish_action` Forcefully finishes an actors current action rather than waiting for it to finish normally.

The example in Figure 1 shows a use of `!!block_on`: the three `!!free` operators are only executed once the actor bound to `?squad` finishes the `!restrain` action. If this operator was left out, the target would be freed immediately, the first squad member freed when the restrain finishes, and the second freed when the complement finishes. The `!!block_on` operator is especially useful for making one squad member wait for another, for example to wait for a command voice over to finish before acting on the order. Finally, `!!finish_action` allows interruption of actions that do not finish (guarding a suspect is such an action in our game).

World State Updating

As we already mentioned above, the updating of the world state that planning is based on is more complicated during plan execution that it is during planning. During planning, all operator effects are immediate and their effects are completely specified by the operator description. During execution, however, some operators take time to execute, and they may fail to achieve their effects, and some of their effects may be internal to the planning system while others are based on sensing the game world. We therefore introduce a three-way distinction amongst the effects of external operators: The predicates they change can be marked to change when the operator starts executing, when the operator stops executing, or as external to plan execution. For example, when the `!restrain` operator begins executing it adds a predicate `(restraining ?squad ?target)` to the world state, which can be used by other plans or the restrain complementer to find the current restrainer. When the action finishes, the operator removes this predicate again. During planning this operator also adds a predicate `(restrained ?target)` to the world state to signal the effect of its application. During execution, however, it does not. Rather, the sensory system will start perceiving the target as restrained once the restrain completes, and add the same predicate. If the restrain fails (say, because enemies appear that should be fought first), the predicate will not be added and the planner may find a similar plan again to perform the restrain.

Squad Behaviors

As sketched in Figure 2 we have used the HTN squad planning and execution system described here to implement several complex team behaviours in a game. Squad members pair up to perform autonomous restraining of suspects as well as investigative actions. When ordered to clear a room, they proceed in pairs, alternatingly advancing once an area is announced as clear, bandaging as needed, calling out doors as they proceed and fighting potential hostiles before continuing. The logic gracefully handles the death of squad members, allowing asymmetric clear behaviours to continue until complete.

Conclusion and Future Work

We have shown an application of HTN planning to generating the behaviours of small teams in computer games. While the plain HTN framework provides a good basis, we have discussed that it is necessary to perform compile-time optimization of the planner as well as timeslicing to achieve suitable performance for a modern real-time video game. Most importantly, however, we have detailed an execution framework for HTN plans that handles plans for several actors, addressing issues of parallelism, synchronization and time-sensitivity as well as sensed operator effects.

While our framework currently handles failed plans in limited cases, such as the case described above where a battle interrupts a restrain plan, it does not handle general operator or sub-plan failure. The first step to adding plan failure handling would be to collect the preconditions for a given step from the whole plan tree, and to check before the step is executed that the preconditions still hold. If they do not, or if the action fails for another reason, the system can re-plan. More graceful failure modes may also be possible, such as re-planning only for failed subtrees. We intend to add this functionality in the near future.

References

Dini, D. M.; van Lent, M.; Carpenter, P.; and Iyer, K. 2005. Building robust planning and execution systesm for virtual worlds. In *Proceedings of the conference for Artificial Intelligence in Digitial Entertainment*.

Epic Games. 2006. Unreal engine 3. `http://www.unrealtechnology.com/html/technology/ue30.shtml`.

Erol, K.; Hendler, J.; and Nau, D. 1994. HTN planning: Complexity and expressivity. In *Proceedings of the American Association for Artificial Intelligence*.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3:251–288.

Gorniak, P., and Roy, D. 2005. Speaking with your sidekick: Understanding situated speech in computer role playing games. In *Proceedings of Artificial Intelligence and Digital Entertainment*.

Gorniak, P., and Roy, D. in press. Situated language understanding as filtering perceived affordances. *Cognitive Science*.

Grosz, B., and Kraus, S. 1996. Collaborative plans for complex group action. *Artificial Intelligence* 86(2):269–357.

Hoang, H.; Lee-Urban, S.; and Munoz-Avila, H. 2005. Hierarchical plan representations for encoding strategic game ai. In *Proceedings of the conference for Artificial Intelligence in Digitial Entertainment*.

Ilghami, O., and Nau, D. 2003. A general approach to synthesize problem-specific planners. Technical report, University of Maryland. CS-TR-4597, UMIACS-TR-2004-40.

Nau, D.; Au, T.; Ilghami, O.; Kuter, U.; Murdock, W.; and Wu, D. 2003. Shop2: An HTN planning system. *Journal of Artificial Intelligence Research*.

Orkin, J. 2005. Agent architecture considerations for real-time planning in games. In *Proceedings of the conference for Artificial Intelligence in Digitial Entertainment*.