AAAI Technical Report WS-11-18

A Real-Time Concurrent Planning and Execution Framework for Automated Story Planning for Games

Eric Cesar E. Vidal, Jr. and Alexander Nareyek
NUS Games Lab
Interactive and Digital Media Institute, National University of Singapore
21 Heng Mui Keng Terrace, Level 2, Singapore 119613
ericvidal@nus.edu.sg, elean@nus.edu.sg

Abstract

This paper presents a framework that facilitates communication between a planning system ("planner") and a plan execution system ("executor") to enable them to run concurrently, with the main emphasis on meeting the real-time requirements of the application domain. While the framework is applicable to general-purpose planning, its features are optimized for the requirements of automated story planning for games—with emphasis on monitoring player-triggered events and handling on-time (re-)generation of story assets such as characters, maps and scenarios. This framework subsumes the traditional interleaved planning-and-execution paradigm used in embedded continual planning systems and generalizes it to a non-embedded context, making the framework ideal for use with contemporary game architectures (e.g., multithreaded game engines, or games with subsystems communicating over a network).

Introduction

Real-time generation of dynamic computer-game experiences is a complex endeavor; as such, A.I. planning is an attractive option for generating such experiences while minimizing complexity. An ever-growing number of games (Orkin 2006; Champandard, Verweij, and Straatman 2009) use *continual planning* techniques to contend with real-time issues—techniques that are also used in other real-world applications ranging from control of single-agent autonomous systems (Chien et al. 1999; McGann et al. 2009) to large-scale multi-agent operations planning (Myers 1998). The basic principle of continual planning is to *interleave* the *creation* of plans with the *execution* of such plans in order to deal with uncertainty in the real world in a timely manner, with the uncertain events determined via *observation* (desJardins et al. 1999). *Real-time continual planning* imposes the additional constraint of having the plan's goals attained within a pre-established response-time deadline (Laplante 2004). Failure of the planning system to meet a deadline has negative consequences ranging from a mere decrease in effectiveness (e.g., the realism of a game character's behavior may decrease) to catastrophic failure (e.g., a mistimed weapon firing may result in real-world property damage).

Copyright © 2011, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

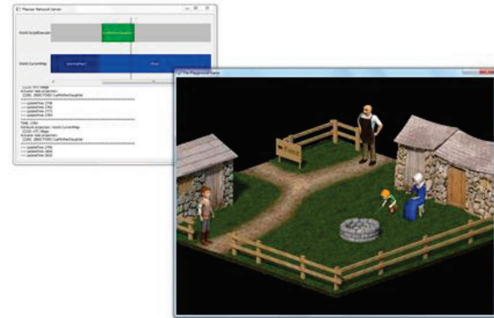


Figure 1. Real-time concurrent planning and execution in action

While some recent systems, particularly the T-REX planner (McGann et al. 2009), explicitly handle the problem of meeting real-time deadlines in domain-independent continual planning, these systems are restricted to domains that permit the planner to be *embedded* within the executor of the plan, allowing strict, controlled interleaving of the planning and execution phases. Such "integrated planners" are not suitable for some problems; an example is the *automated story planning problem*, a generalization of previous attempts for A.I. planning within games (Nareyek et al. 2009). This study goes down to the level of the human brain's reward system to determine the story's elements, i.e., by detecting player *motivations* and giving appropriate *rewards* (as well as *cues* to further increase motivation to achieve said rewards), all at exact times. For this problem, the main planning system is separated from the game engine (see Figure 1) in order to achieve application independence and general applicability with many game types.

Ideally, a general planning system should be applicable to all kinds of domains, including the automated story planning domain. Embedded planner architectures (with

strictly-interleaving planning and execution phases) are ill-fitted for domains with separate planning and execution subsystems; thus, a better approach for a general real-time planning system is a *concurrent architecture* where the planner and executor can run independently, communicating only as necessary. However, a concurrent architecture exacerbates *latency problems* that were either non-existent or trivial in the embedded approach. These problems may interfere with the real-time attainment of planning deadlines, e.g., when a player exits the current game map, the next map may not be shown to the player on time due to latency in the receipt of the map-exit signal, or latency in sending the details of the next map; either way, disrupting the story's continuity. Thus, this necessitates plan execution and monitoring algorithms that can specifically deal with these latency problems.

In this paper, a general framework is proposed to address this problem of *real-time concurrent planning and execution*. This framework consists of the *architecture* encapsulating the communication aspects between the planner and the executor to allow the two subsystems to run independently, and the *interface guidelines and algorithms* for how the planner and the executor subsystems communicate through this architecture.

In addition, real-world domains typically have specific real-time requirements from the framework—the automated story planning domain, for instance, requires real-time support for dealing with player actions and background generation of story assets such as maps and characters. Thus, the framework supports *planning extensions* to handle such domain requirements in a general manner. This paper briefly discusses two such extensions—*rule monitoring* to deal with external events with a level of non-determinism (e.g., if certain actions expected to be done by the player do not occur) and *object anchoring* to deal with connecting planner-conceived objects to actual objects and reconnecting them to different objects if needed (e.g., creating a castle map during initial deliberation, then subsequently replacing it with a village map if it is later deemed to be a better experience for the player).

The rest of the paper is structured as follows: The next section reviews existing work in interleaved planning and execution systems, focusing on how real-time aspects are typically handled (and how previous methods are lacking). This is followed by a case study, the Automated Story Planning domain, which requires a concurrent model of planning and execution in place of the interleaved model. Next, the real-time concurrent planning and execution framework, as well as the two extensions (rule monitoring and object anchoring), are presented and discussed. The paper concludes with a preliminary evaluation of the ongoing implementation work and a list of future work.

Background

Although there is previous research on concurrent planning and execution, particularly for robotics-based domains such as (Simmons 1992) and (Miura and Shirai 1998), the focus of these work is on achieving better performance over sequential planning and execution, and not necessarily on meeting the explicit real-time *deadlines* of a problem. With this in mind, the first subsection focuses on reviewing continual planning systems that explicitly handle execution of actions in real-time. The second subsection tackles the related problem of execution *monitoring* for these systems, again identifying the real-time aspects of the problem.

Real-time Execution in Continual Planners

Few continual planning systems actually consider the real-time aspect of planning problems explicitly. For example, the planning system in the game F.E.A.R. (Orkin 2006) is considerably simplified by omitting timings from the domain specification of actions, leaving the animation system to handle timing details; this system can therefore meet only *soft* real-time deadlines (i.e., missing a deadline is not a system-breaking concern); it cannot plan for an action to be executed at an exact given time. Façade (Mateas and Stern 2003), on the other hand, explicitly encodes exact timings for such actions (such as "nod head for 1sec" or "wait for player reaction for 2secs"), but uses an even more simplified paradigm that is more reactive than deliberative, i.e., it only does temporal scheduling of those actions within a single story "beat"; the goals of the beat and the options for succeeding beats are predetermined, not planned.

More general planning techniques are available in the CASPER planner used in embedded spacecraft systems (Chien et al. 1999), which consider *durative actions* and an *anytime* approach to planning via the use of local search and iterative repair. Particularly, the approach entails the use of anytime conflict-resolution methods which ensure that a valid partial plan is returned given a time constraint for deliberation, e.g., "steer the vehicle out of the asteroid's path within one second", while also ensuring the success of the long-term plan, e.g., "take photos of all observable asteroids". However, no consideration is given to *ensuring the timely execution* of the planned actions, i.e., there is no regard for *latency in dispatching actions* from the system that planned the action to the system that executes it.

The T-REX system used by the MBARI Autonomous Underwater Vehicles (McGann et al. 2009) considers this action dispatch latency. T-REX is a multi-layered system of self-contained planning subsystems (dubbed 'teleo-reactors'), each with a *deliberation latency* (how long can a reactor deliberate for one task) and *planning horizon* (how long is that reactor active). The lowest-level reactor, dubbed the 'executive', essentially performs the execution of primitive plan actions. T-REX, unlike previous systems, makes *explicit real-time guarantees* in action execution: an action to be dispatched between two reactors will execute on time as long as it is dispatched within the time window bounded by the deliberation latency and the planning horizon of the reactor executing the action; actions should not be planned outside of this time window.

However, the method works around timing difficulties between reactors by assuming a common real-time clock shared across the entire system to synchronize action dis-

patching. This effectively assumes an *embedded* system approach, where the subsystem that is planning an action is in the same physical system as the subsystem that is executing the action. Consequently, the synchronization discussion in (McGann et al. 2009) does not account for inter-system *transmission latency*, i.e., how long an action takes to be transferred from one system to another if the two systems are independently-running and message transmission is not instantaneous.

Real-time Monitoring in Continual Planners

Another aspect of continual planning systems is the use of *sensors* to monitor data about unknowns in the world into the planner (i.e., external events, or data that can only be partially observed at any given time), in order for the continual planner to adapt to unpredictable changes in its world. Some story systems that accomplish real-time plan execution, e.g., Darshak (Jhala and Young 2010) and the Merchant of Venice system (Porteous et al. 2011), only have limited sensing support (e.g., only for things such as the current position/point-of-view of the player) and are unconcerned about other events, e.g., player actions.

CASPER (Chien et al. 1999) and T-REX (McGann et al. 2009) support sensors more extensively (e.g., vehicle position and orientation, object identification, fuel consumption etc.), but typically assume that these sensors continuously and passively feed data into the deliberating system. However, continuous passive monitoring may be costly (e.g., the act of sensing may use more battery power in a robot, or too many sensor inputs may overwhelm the planner in terms of attention), and in some cases impossible (e.g., the robot cannot know what is behind a door unless, for example, the said robot opens the door).

Planning systems in other real-time games work around the problem of excessive continuous monitoring by either restricting the planning representation of the game world into a small state array which can be easily sensed and updated (Orkin 2006), or using a daemon architecture to *filter* world state for use by a planner (Champandard, Verweij, and Straatman 2009). These are not general solutions, however, and inevitably a planner that needs to manage huge amounts of partially-observable information will need to use *active monitoring techniques*, where the deliberating system specifically requests for observations as needed via *sensing actions*. Such active sensing can be used for multiple purposes, e.g., to test for the *availability* of information needed for decision making or to track *expectations* after an action has been executed (Myers 1998).

Unfortunately, sensing actions are subject to the same latency issues as other actions, and therefore can affect the timeliness of system *reaction* towards newly sensed information. An example to ground this is the following scenario: If a game interleaves planning with monitoring of, say, player clicks, the game A.I. can only react to a player click on the next planning cycle. Note that in games it is fairly common to have a planning cycle rate that is slower than actual game rate, e.g., 5Hz planning rate and ~60Hz game rate (Champandard, Verweij, and Straatman 2009). If the game's planner wishes to accurately predict the time of the next player action (e.g., to synchronize A.I. character behavior with the player's), the planner should be aware that a "current" click actually happened in the past and must account for the delay accordingly. Although previous real-time planners account for such delays at least in the passive sensing case, e.g., T-REX has a mechanism to dispatch state variable observations from the executing reactor back to the planning reactor, inserting them at their proper times in the past (Py, Rajan, and McGann 2010), these previous systems do not account for separated planning and execution subsystems, where sensing actions must be planned and thus incur inter-system transmission latency (i.e., planning reaction after the dispatch of a sensing action is further delayed by the round-trip latency of the connection between the planner and executor).

Case Study: Automated Story Planning

Most of the aforementioned real-time planning systems use embedded architectures that strictly interleave planning and execution, but as described earlier, this design is ill-fitting for systems where the planner and executor are independent. Such a case is described in this section.

The Automated Story Planning for Games project (Nareyek et al. 2009) is an on-going research project intending to provide computer-generated stories in real-time. The basic goal of the system is to optimize the *entertainment experience* of a player. The player's *motivation* is represented as a numeric value that changes over time (see top half of Figure 2), and *cue* and *reward actions* are executed at the exact moments to create a build-up or decrease of motivation, respectively (see bottom half of Figure 2). The resulting "motivation curve" is optimized to follow a pattern of crests and troughs (representing episodic points in the story), with the highest peak and lowest dip (the ultimate reward) near the end of the game, all in all translating to a good entertainment experience.

Cue and reward actions are actually game content pieces (e.g., an animation depicting the princess kissing the player, or an animation of townsfolk celebrating the player's success against a terrorizing monster) which are authored in the usual way by game developers; the main difference between this and other game story generation methods is that these content pieces are automatically *planned* over time using a goal-based planner (with the goal specified as "generate an optimal player motivation curve over the en-

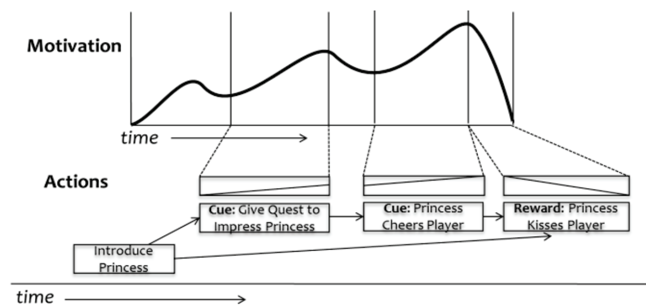


Figure 2. Player motivation curve vis-à-vis planned actions

tire run of the game"), instead of having the game designer dictate if (and when) an animation will be played. To actually show these cues or rewards, however, certain preconditions must be met, such as "there has to be a princess in the game first" or "there has to be a castle where the princess lives" (asset generation), or "there needs to be a competition quest that the princess will be witnessing, so that the player is motivated to impress the princess" (scene generation). These ancillary actions also need to be planned along with the planning of cues or rewards.

The Automated Story Planning system can be characterized as a *firm* (as opposed to soft) real-time system. The exact timing of the inflections in the motivation curve are essential to the player experience model—the player's motivation is severely affected with incorrect timing of rewards (e.g., if the romance motivation is important to the player and the player receives the kiss from the princess too early, there would actually be *no* motivation to see the game to the end even if there are other rewards coming). Furthermore, a delay or unintended advance in the execution of any of these actions will change the projected motivation curve in the future, and it may not be possible to correct the motivation curve at a later point, i.e., failures may propagate and future goals might not be attained at all.

In terms of architecture (see Figure 3), the planning system of the game (called the *experience manager*) is intended to be a general system where potentially any game engine (in the figure, called the *scene manager*) can be connected. The reasons for separating the planning system from the game engine into concurrently-running modules are as follows:

- With a separate planner and game engine architecture, an execution module can be built into any stock game engine to receive and execute partial plans from the experience manager, making the planner (and domain) design impervious to game-specific details such as engine/language platform.
- Individual cue or reward actions may be in the form of complex *scripts* executed within the game engine (e.g., a game 'conversation' can be broken down into a series of dialog boxes presented in sequence, or two game characters interacting with each other may require many atomic animation actions playing concurrently). Forcing an interleaved planning and execution architecture on such a game engine greatly complicates implementation for these kinds of systems with already many concurrent tasks executing, many of which are tasks that are external to the main story plan (e.g., animation, GUI or physics systems within the engine).

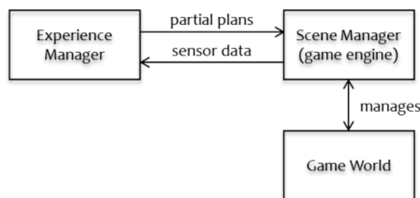


Figure 3. Automated Story Planning architecture

- A concurrent architecture also greatly simplifies debugging, because the experience manager can, for example, be run in a physically separate system (e.g., with the two systems connected via TCP sockets) and the planner's run can be traced and analyzed separately from the game engine's other debugging output.

The architecture proposed here is not unlike other planning-based story generation architectures such as MIMESIS (Young et al. 2004), where a partial-ordered planner is paired with an external game engine (for proof of concept, the Unreal engine was used). However, in MIMESIS, planning is done only partially online, with real-time uncertainties in the world handled via specialized *contingent-planning*-like constructs (either by *intervention*, with the game preventing the player from doing unwanted actions, or *accommodation*, with the plan containing conditional constructs to adapt to the player's plan-breaking actions; full re-planning is triggered when neither of these are possible). In the Automated Story Planning project's case, the story planner is fully on-line; the story planner is meant to automatically adapt to player preferences (e.g., if the system detects that the player's preference tends towards blonde women, a princess asset with blonde hair can be generated) or in-game player actions (e.g., if the player tends to interact with the village girl more instead of the princess, the future cues and rewards are adjusted or even completely replaced in the background to reflect a change in love interest), all in real-time as the planning system tries to optimize the player motivation curve. The real-time aspect of this problem, coupled with the architectural complexity of separating the planner and the plan executor, presents an interesting challenge that is resolved by the proposed framework (discussed in the next section).

General Framework Design

The proposed real-time concurrent planning and execution framework assumes two existing modules: an *anytime local-search planner* module capable of producing partial grounded plans within a fixed time interval (e.g., 0.2 seconds), and an *executor* module capable of executing individual plan actions and reporting monitored sensor values.

For the anytime local-search planner module, we assume the planning model used in the Crackpot planning system (Vidal and Nareyek 2010; Kumar and Nareyek 2011), a descendant of the EXCALIBUR planning system (Nareyek 2001) that is similar in strategy to CASPER (Chien et al. 1999). A complete discussion of the planning model is found in the aforementioned references, but is briefly explained here using an example domain where a person needs to eat an apple that is not in his hand (see Figure 4): Plan *actions* (such as **EatApple**) consist of *conditions* on object *attributes* for the action to successfully execute (e.g., ***apple.position*** must be "inHand" at time *t*, where *t* is the start time of the action), *contributions* to attributes when the action is executed (e.g., ***person.hunger*** decreases over the duration of the action), and *action tasks* which

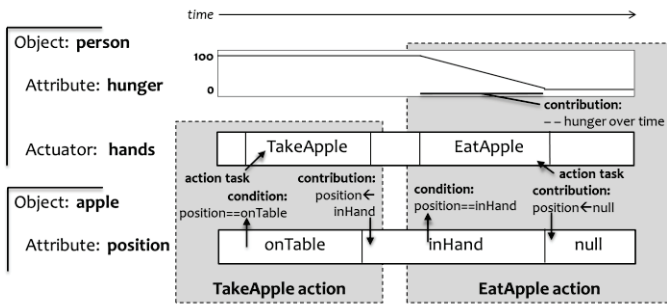


Figure 4. Crackpot plan representation

take up resource on object *actuators* to actually perform or actuate the actions (e.g., ***person.hands*** is used for the duration of the action). Not shown in the figure are *read-ins* from attribute values at certain time points to determine action variables (e.g., the action's duration may come from the size of the apple). A current plan is *iteratively repaired* by determining *costs* in a plan and selecting one of these costs for repair per planning iteration; these costs are either unmet *goals* on attributes' value projections over time (e.g., "***person.hunger*** must be 0 at time 500"), unmet conditions (e.g., if ***apple.position*** is "onTable" at the start of the **EatApple** action), overlaps on actuator usage (e.g., ***person.hands*** cannot be used for taking an apple and eating it at the exact same time) or inconsistent structural constraints within an action (e.g., the action task's duration should be the same as the contribution's duration; in Crackpot, such constraints are supported via *action-component relations*). Cost repairs are selected using *heuristics* built into the planner, which may involve spawning new actions, moving actions, changing action parameters, or spawning new objects (e,g,, **TakeApple** is added to fix the cost introduced by **EatApple**'s condition). After a number of repair iterations, a portion of the plan is sent to the executor while the planner resolves other costs.

The real-time planning and execution framework itself has a base architecture (see Figure 5) to facilitate communication between the planner and executor modules. The framework provides *interface wrappers* local to each module such that the underlying messaging protocol is hidden under a layer of abstraction. This concurrent design permits a wide variety of implementations, e.g., via sockets or message queues provided by the operating system. (In fact, an embedded system implementation with interleaved calls to the planner and executor is still possible; in that case, communication between planner and executor is trivialized and is assumed to have zero latency.)

The planner side's processing logic works in a sequential manner—the planner and the planner-interface modules actually *interleave* in processing. This design allows a regular interleaving-aware continual planner to work with the architecture with only minor modification—the intent is for the planner-interface to mimic the behavior of a plan executor that the planner can directly talk to. On the other hand, the executor and executor-interface may operate in a concurrent manner, with the commands and queries made by the executor-interface not required to occur in lock-step with the regular processing of the executor.

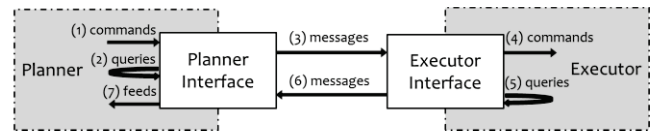
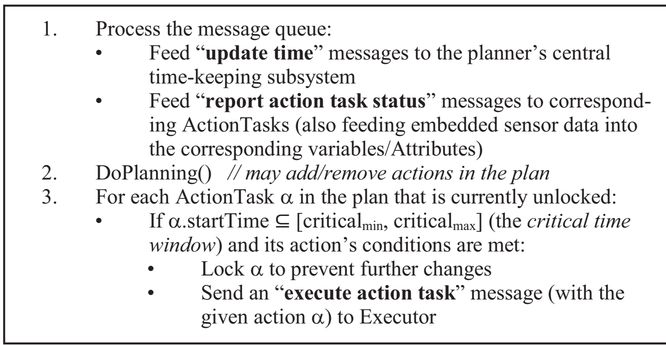


Figure 5. Architectural design of the framework

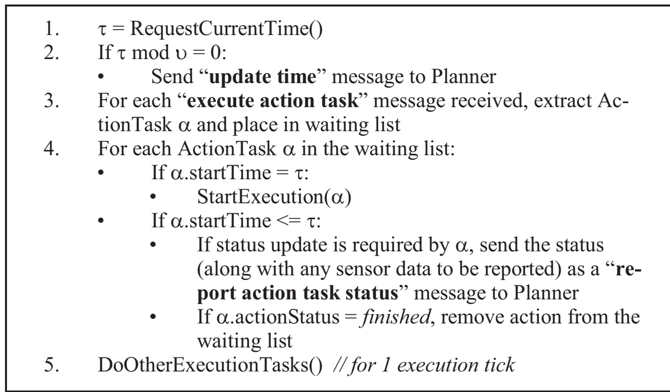
For the purposes of discussion, data transferred between the two modules are referred to as *messages*. To illustrate the intended interface between the two modules, a typical action dispatch goes through the following control flow, illustrated in Figure 5 and formalized in the algorithms shown in Figure 6 and Figure 7:

1. Each time a planner creates or deletes an action task, the planner sends a *command* to the planner-interface to add/remove the task from the planner-interface's tracking list.
2. The planner-interface, at the end of a planning cycle (e.g., after every 0.2 secs), *queries* the planner about the action tasks in its list for execution information ("what time should action task α start executing?"; "are the conditions of α's action currently met?").
3. For action tasks whose start time is close to current time (this "critical time window" is discussed in the next subsection) and whose conditions are met, the planner-interface sends a *message* to the executor-interface containing the execution details of the action task. Note that this also entails *locking* of the action task before sending, i.e., the planner-interface tells the planner that α's parameters may no longer be changed.
4. The executor-interface receives the message and translates it into a local *command* understood by the executor. A message is sent such that it is typically received ahead of time (due to uncertainty in message arrival time), so the commands are placed in a waiting list within the executor-interface (implementation-wise, this should be sorted according to time for efficiency), and the command is actually executed at the exact time for execution.
5. The executor-interface *queries* the status of all executing action tasks (e.g., whether the task has executed at all, or in the case of sensing actions, whenever sensor data is available).
6. If any status updates are available for an action task, the executor-interface transmits these updates back to the planner-interface via *messages*.
7. The planner-interface receives the incoming message(s) and then *feeds* the data back into the planner (e.g., "α is currently executing"). In the case of a sensing action task, the sensor values in the message are fed into the relevant attributes/variables as *observations* (e.g., "***player.location***=**village** @ time [236, 336)" or "α***.action.duration***+=10"). These observations override any projected information already stored in the attributes, allowing the planner to re-project attributes and re-plan accordingly.



1. Process the message queue:
   - Feed "**update time**" messages to the planner's central time-keeping subsystem
   - Feed "**report action task status**" messages to corresponding ActionTasks (also feeding embedded sensor data into the corresponding variables/Attributes)
2. DoPlanning() *// may add/remove actions in the plan*
3. For each ActionTask α in the plan that is currently unlocked:
   - If α.startTime ⊆ [$critical_{min}$, $critical_{max}$] (the *critical time window*) and its action's conditions are met:
     - Lock α to prevent further changes
     - Send an "**execute action task**" message (with the given action α) to Executor

Figure 6. Planner-interface algorithm



1. τ = RequestCurrentTime()
2. If τ mod υ = 0:
   - Send "**update time**" message to Planner
3. For each "**execute action task**" message received, extract ActionTask α and place in waiting list
4. For each ActionTask α in the waiting list:
   - If α.startTime = τ:
     - StartExecution(α)
   - If α.startTime <= τ:
     - If status update is required by α, send the status (along with any sensor data to be reported) as a "**report action task status**" message to Planner
     - If α.actionStatus = *finished*, remove action from the waiting list
5. DoOtherExecutionTasks() *// for 1 execution tick*

Figure 7. Executor-interface algorithm

Note that active monitoring is the primary method of sensing used in the framework; all sensing is done via action tasks. An exception is the *sensing of current time*, which is assumed internal to the executor (for discussion, an execution time unit is referred to as a *tick*) and is fed continuously to the planner, eliminating request overheads. *Continuous subscription* to a particular sensor (which may be used for, say, intensive monitoring of the position of an in-game character that is important to the current scene) may be made by modeling the subscription as a long-running action task which returns sensor values at regular intervals.

Timing Control

The *critical time window* and other timing variables mentioned in the algorithms bear some discussion. The framework assumes that action tasks are sent *ahead of time* to combat the effects of various latencies. These latencies are formalized via these *timing control variables*:

1. **Planning window** (ω) – the worst case number of ticks an executor will progress during a full planning cycle (which may be composed of thousands of local-search iterations). For example, for a 60Hz executor, ω = 12 yields an effective planning window of 200ms.
2. **Execution update window** (υ) – the interval between the executor's current-time updates in number of ticks. Note that if this is set equal to the planning window ω, this will not guarantee that the planner will get 1 time update at every planning iteration, owing to uncertainty in the arrival of current time updates. Thus, this value should be set lower, e.g., if ω=12, υ may be set to 6 (i.e., 100ms).
3. **Message-passing latency** (μ) – the worst case number of ticks before a message is completely transmitted from the planner to the executor, or vice versa. In an embedded system with tight cooperative scheduling of planner and executor (i.e., interleaving), μ = 0. If OS-controlled pre-emptive scheduling is used, however, this value must be set to the timing resolution of the OS's context switching. A full analysis of the proper value for this variable requires statistical modeling, e.g., using a Poisson arrival model (Laplante 2004).

Figure 8 is a visual depiction of these timing variables, illustrating how planning occurs asynchronously with execution (as opposed to interleaving). In the best case, an executor update (e.g., a current time message) will incur a delay of μ if it arrives exactly on time for the next planning iteration (see top half of Figure 8); however, in the worst case, the messa ge will be queued until the next planning cycle and its processing will be delayed by at most υ + μ (see bottom half of Figure 8). If the planner then requests for immediate execution of an action to react to current world state (e.g., if the latest observation includes ***player. location*** = **village**, and an impending **KissPlayer** reward action requires the princess to be at the same location as the player, the planner might insert a **MoveCharacter** action moving the princess to the village), the arrival of the command on the executor will be delayed by μ in the best case (command is executed immediately) and by μ + 1 in the worst case (command is queued for the next execution tick). Since actions are only dispatched at the *end* of the planning cycle, the known current time in the planner side would have advanced by at most ω from the last reported time of the executor. Therefore, planner heuristics to insert new actions into the plan must ensure that the start time of an action intersects with a *critical time window*, the *minimum* of which is given by the following formula:

$$critical_{min} = \tau + \upsilon + 2\mu + \omega + 1$$

where τ is the last reported current time. Since the planner is not able to change the temporal placement of actions once it has been dispatched to the executor, dispatching of an action far ahead into the future must be deferred to allow re-planning for contingent events. This means that the

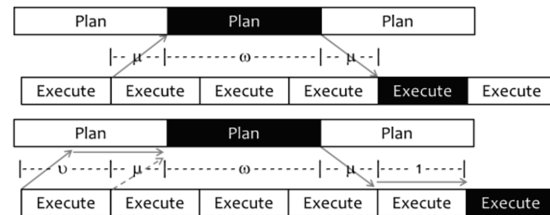


Figure 8. Various latencies during concurrent planning and execution: best case (top) and worst case (bottom)

maximum of the critical window is the start time of next planning cycle, thus given by the following formula:

$$critical_{max} = \tau + \upsilon + 2\mu + 2\omega$$

Case Examples of Framework Extensions: Usage with the Automated Story Planning Domain

Planner *extensions* are sometimes necessary to handle certain requirements of planning problems. Discussed here are two such requirements in the Automated Story Planning domain and how the general framework is extended to handle these requirements: monitoring uncertain player behavior, and handling real-time generation of story assets.

Monitoring the Rules of Player Behavior

Apart from the actions that the planner may enact on the virtual game world, the game itself may introduce events that modify the world in response to the changing world state. Such *world dynamics* may be modeled via the use of deterministic *rules* (Nareyek and Sandholm 2003); a rule is essentially defined like a plan action but is triggered automatically the moment its conditions are fulfilled (instead of being decided to be put into the plan), e.g., when a ball is released in mid-air, a "gravity rule" is always triggered such that the ball falls to the ground. These rules enable the planner to reason about future world states, e.g., a computer-generated soccer player may kick the ball at the exact moment the ball hits the ground.

However, the existence of a (non-deterministic) player presents a challenge that needs to be corrected for in real-time. For example, the planner may determine initially that the player will have the best experience by giving him a quest. The domain may encode a rule where if a villager tells the player of the existence of a precious diamond in the forest, the player will, as a result, move towards the forest. This, however, opens two contingent possibilities apart from the expected outcome: the player may either delay his/her movement towards the forest, or may not go to the forest at all (i.e., the planner's prediction is wrong).

Thus, the framework is extended with a mechanism for *rule monitoring*—rules are outfitted with special *rule tasks*, whose purpose is to monitor critical attributes for the expected changes to the world state (see Figure 9). These rule tasks are specified with *deadlines* such that if the expected attribute change did not happen, either the duration of the rule is extended, or the rule is invalidated (depending on modeling needs). In essence, these rule tasks function similarly to regular action tasks, and the framework can thus dispatch and monitor them in the same manner.

Object Anchoring for Asset Generation

It is implausible for some story assets (e.g., a castle) to be present in the world at one moment and disappear in the next, or for immovable objects to change position, etc. While a story generator may introduce objects at any time, it may not change parameters of these objects as soon as these parameters are manifested, e.g., if the princess's hair is initially blonde, it (usually) cannot be changed when the player has seen it. However, some parameters should still be changeable by the planner *until* they are manifested to the player (to allow the local-search planner to make on-time optimizations while earlier parts of the story are still being presented). This necessitates a two-step process for generating objects (illustrated in Figure 10):

- *Anchoring* is the process by which a planner-conceived object (e.g., a new castle map to satisfy the requirements of a princess-presentation scene) is connected to a real-world (virtual) object. In the case of the story planning domain, an anchoring action tells the executor side to create the necessary assets for this object, along with any requested parameters (e.g., "create a princess with blonde hair"). An object may be re-anchored if necessary (e.g., if the player decides to forego the princess quest in favor of a save-the-village quest, the castle map may be re-anchored to a village map).
- *Commitment* is the process of "experiencing" a game asset (e.g., the player enters the village map). As these are player-initiated, commitment should be encoded in the domain as rules (as mentioned in the previous subsection), with a condition that the object needs to be anchored first, plus other triggering conditions, e.g., "player is on the way to the map". A committed object cannot be de-anchored, e.g., an already-experienced map cannot disappear.

This allows for on-time re-generation of game assets as story planning goals change. The pattern of anchoring and commitment is repeated for all so-called "constructible" objects in the domain (and thus can be auto-generated by a preprocessor on the domain level). Additional conditions for construction may also be placed on the anchor action or commit rule depending on authorial constraints (e.g., "a

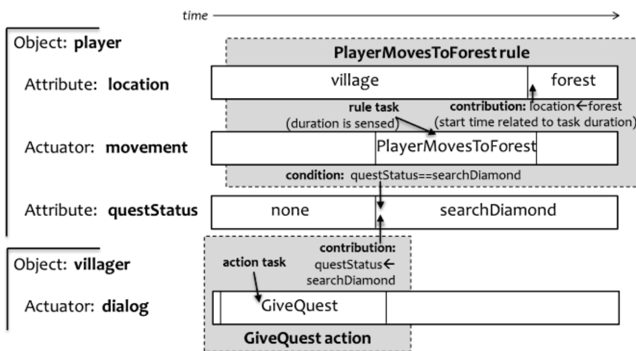


Figure 9. Rule tasks for monitoring on-going rules

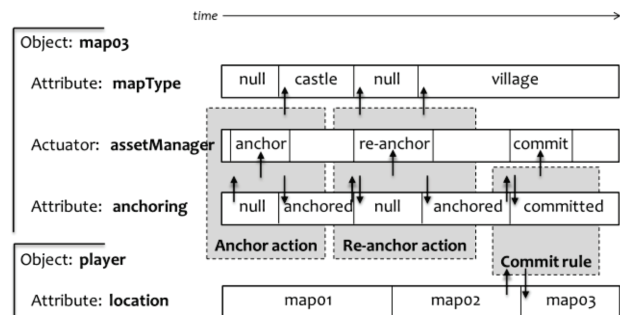


Figure 10. Anchoring example; arrows pointing out from attribute projections are conditions, other arrows are contributions/tasks

princess can only be constructed if ***story.princessExists*** = *false*" where ***story*** is a non-constructible, pre-existing object). For optimization, a special planning heuristic may be used to minimize expensive re-anchoring actions (i.e., there should only be one re-anchoring past current time); this heuristic, in turn, will use the framework's critical time window to decide which re-anchoring tasks to remove.

Preliminary Evaluation

A prototype implementation is built using the Crackpot planner and a game based on PlayFirst's Playground SDK as proof-of-concept of the framework's applicability to off-the-shelf multithreaded game engines. Figure 1 shows the prototype game and planner running concurrently.

While the game is in an alpha stage and the story planning domain model is currently in development, the system's capability for real-time planning and execution of very simple plans is already promising, with single plan actions executing on-time even with a fast execution tick rate (60Hz) greatly exceeding planning rate (1Hz) and with conservative values for the timing control variables (ω = 60, υ = 12 and μ = 1; i.e., 1Hz planning window, 5Hz execution update window and 16.7ms message passing latency). Further evaluation of the framework will involve reliability tests using other domains with measurable outcomes (e.g., real-time variants of logistics or job-shop-scheduling domains) and analytical modeling of the system's reliability such as in (Chen, Bastani, and Tsao 1995).

Conclusion

The proposed framework realizes *concurrent planning and execution* with *real-time guarantees* for action execution (assuming proper modeling of latency via implementation-specified timing control variables). This framework is a generalization of interleaved planning and execution, allowing real-time planning applications with complex architectures (e.g., Automated Story Planning) to be developed.

Future work includes full development of the Automated Story Planning prototype game, which will likely expose additional real-time sensing and execution issues, and the framework may require special heuristics to produce better plans given real-time constraints. Development of framework extensions for other real-world domains (e.g., action negotiation between planner and executor, planning-to-sense, etc.) is also forthcoming. Lastly, the framework, with some modification, may be further extended to planning problems utilizing or even spawning multiple execution clients, e.g., enlist more robots to finish a job faster.

Acknowledgments

This work was supported by the Singapore National Research Foundation's Interactive Digital Media R&D Program under research grant NRF2007IDM-IDM002-051.

References

Champandard, A.; Verweij, T.; and Straatman, R. 2009. Killzone 2 Multiplayer Bots. In *Game AI Conference,* Paris.

Chen, I.; Bastani, F. B.; and Tsao, T. 1995. On the Reliability of AI Planning Software in Real-Time Applications. *IEEE Transactions on Knowledge and Data Engineering* 7(1): 4-13.

Chien, S.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 1999. Integrated Planning and Execution for Autonomous Spacecraft. In *Proceedings of the 1999 Aerospace Conference,* Pasadena, CA.

desJardins, M. E.; Durfee, E. H.; Ortiz Jr., C. L.; and Wolverton, M. J. 1999. A Survey of Research in Distributed, Continual Planning. *AI Magazine* 20(4): 14-22.

Jhala, A., and Young, R. M. 2010. Cinematic Visual Discourse: Representation, Generation, and Evaluation. *IEEE Transactions on Computational Intelligence and AI in Games* 2: 69-81.

Kumar, A., and Nareyek, A. 2011. An Extensible Planning Architecture for Configurable Rich-World Actions. In *AAAI 2011 Workshop on Generalized Planning,* San Francisco, California.

Laplante, P. A. 2004. *Real-Time Systems Design and Analysis, 3rd Edition.* Wiley-IEEE Press.

Mateas, M., and Stern, A. 2003. Façade: An Experiment in Building a Fully-Realized Interactive Drama. In *Game Developers Conference,* San Jose, CA.

McGann, C.; Py, F.; Rajan, K.; and Olaya, A. G. 2009. Integrated Planning and Execution for Robotic Exploration. In *International Workshop on Hybrid Control of Autonomous Systems, IJCAI 2009,* Pasadena, CA.

Miura, J., and Shirai, Y. 1998. Scheduling parallel execution of planning and action for a mobile robot considering planning cost and vision uncertainty. In *Proceedings of the International Conference on Intelligent Robots and Systems,* Victoria, BC, Canada.

Myers, K. L. 1998. Towards a Framework for Continuous Planning and Execution. In *Proceedings of the AAAI Fall Symposium on Distributed Continual Planning,* AAAI.

Nareyek, A. 2001. *Constraint-Based Agents: An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds (LNAI 2062).* Springer.

Nareyek, A.; Bostan, B.; Vidal Jr., E. C.; Behpour, S.; Koh, S. M.; Wang, H.; Malik, O. N.; Siew, Z. X.; Agarwal, A.; Manunethi, S.; and Wong, M. T. 2009. Automated Storyplanning for Games: Concept Details and Solution Approaches. Technical report (UNPUBLISHED), NUS Games Lab, National University of Singapore, Singapore.

Nareyek, A., and Sandholm, T. 2003. Planning in Dynamic Worlds: More Than External Events. In *IJCAI-03 Workshop on Agents and Automated Reasoning,* Acapulco, Mexico.

Orkin, J. 2006. Three States and a Plan: The A.I. of F.E.A.R. In *Proceedings of the Game Developer's Conference (GDC),* available at http://web.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf.

Porteous, J.; Teutenberg, J.; Charles, F.; and Cavazza, M. 2011. Controlling Narrative Time in Interactive Storytelling. In *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS),* Taipei, Taiwan.

Py, F.; Rajan, K.; and McGann, C. 2010. A Systematic Agent Framework for Situated Autonomous Systems. In *9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS),* Toronto, Canada.

Simmons, R. 1992. Concurrent Planning and Execution for Autonomous Robots. *IEEE Control Systems* 12(1): 46-50.

Vidal, E. C., and Nareyek, A. 2010. An XML-based Forward-Compatible Framework for Planning System Extensions and Domain Problem Specification. In *ICAPS 2010 Workshop on Knowledge Engineering for Planning and Scheduling,* Toronto, Canada.

Young, R. M.; Riedl, M. O.; Branly, M.; Jhala, A.; Martin, R. J.; and Saretto, C. J. 2004. An architecture for integrating plan-based behavior generation with interactive game environments. *Journal of Game Development* 1.