Unified Planning and Execution for Autonomous Software Repair

Richard Levinson

Attention Control Systems, Inc.
650 Castro Street, Suite 120, PMB 197
Mountain View, CA 94041
rich@brainaid.com

Abstract

This paper addresses the need for more flexible autonomous systems that detect and correct software failures at runtime. We present new methods for integrated planning and execution that enable runtime verification and repair for software failures, and explore the related issue of integrated procedural and declarative action representations.

We present a library for embedding declarative planning methods within procedural C++ code. The library provides an interface to supervisory processes, which monitor software execution and provide last-resort error recovery after pre-programmed error handlers fail. The library provides an interface to search and temporal constraint engines maintained by the meta-processes. The planner and controller use the same procedural representation in order to share context (computational state) between planning and execution.

Motivation

Limited Autonomy. Today's autonomous systems provide more coverage for hardware failures than software failures. If they cannot represent and reason about software failures, they are doomed to blind spots and will have limited autonomy. There are several reasons why software failure cannot be avoided including: limited time and information at design time, limited time and resources for running test cases, changing operating conditions and changing mission requirements. Our goal is to include more software within the scope of recovery and develop autonomous systems that repair their own software.

Limited Architectures. Currently, most system software is outside the scope of plan-based recovery because it is not written in the planner's modeling language. To increase model scope, the planner and controller must share computational state information about failure and repair contexts. This is challenging because integrated planning and execution traditionally involves translating between planners that use declarative languages and controllers that use procedural languages. Major problems caused by this *language barrier* include:

Redundant & Low Fidelity Models. There is a need to develop and maintain a declarative model of the execution system using a planner's modeling language. The planner's model is redundant with the execution "model" (i.e. the software). The planner's model of software is low fidelity since much of the computational state is hidden in a black box model of software. The redundant planning model may not match the actual execution software, and it is difficult to maintain the model and the code in parallel.

Information loss is nearly guaranteed when translating between the controller's procedural language and planner's declarative language, which are typically developed independently and optimized for different purposes. This information loss reduces the planner's understanding about execution context, and vice versa.

System complexity is increased because of the need to translate between the two languages. There is an increased need to develop ad hoc translators between the two languages, and a lot of the integration effort is devoted towards accommodating specific language differences.

Architecture Overview

Our approach to increasing the scope of failure recovery and removing the language barrier is called the **Program Planning and Execution Language (Propel)**. Propel provides a library for integration between C++ and supervisory processes that detect and correct software failures. The library includes new methods for integrating search and temporal constraints within C++ applications.

We also present new methods for integrated planning and execution with improved computational context switching and sharing capabilities compared with existing methods. These new methods allow us to increase the scope of error handling to include the system's software infrastructure. Additionally, the planner and controller's action representations are unified so more context information can be preserved during transitions between execution and planning.

Figure 1 shows the three different process levels within Propel: The *Application*, *Supervisor*, and *Executive* levels. The Application level contains all of the domain-specific processes. The Executive and Supervisor levels are meta-processes (they monitor and manipulate the application processes) in order to detect and correct software failures.

The **Application level** executes C++ code extended with the Propel Library Interface to meta-level supervisory processes. The code contains declarative statements that interface to search and temporal constraint engines.

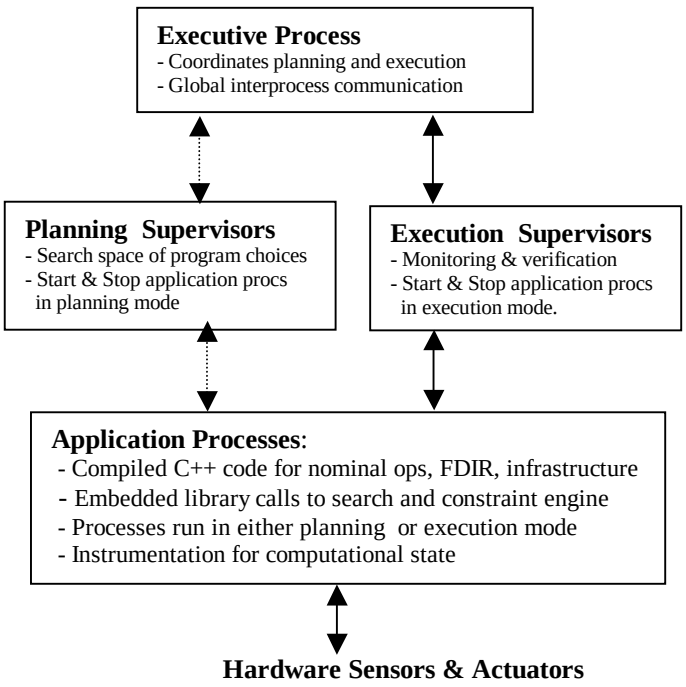

Figure 1: Propel's three levels of processes: *Application, Supervisor, and Executive.*

Choice points embedded in the code describe alternative operations and resources, and define a space of program variations. The Application level has information only about its own execution environment, like any other C++ process. The application level sends status messages, computational state information, and queries to the meta-level supervisors, and are started and stopped by the supervisors.

The **Supervisor level** processes perform runtime monitoring,, verification, and recovery of application level processes. They manage search spaces of application level processes. One planning supervisor and one execution supervisor is created for each parallel process in the application. The planner provides a last-resort failure handler after all application-level fault protection has failed. The term *"planner*" refers to a Planning Supervisor and the term "*controller"* to refer to an Execution Supervisor.

The Planners and controllers start, monitor, and stop the same compiled application-level code in the same environment. The difference is that the planners run the application level code with a _PLANNING_ flag true but the controllers run the code with that flag false. Application code accesses that flag to determine if it is running in planning (simulation) or execution mode.

Application code may execute different branches depending on the status of this flag. Subroutines which send out physical actuator commands may simulate the commands when the planning flag is true instead of sending out actual actuator commands. The only procedures that require simulation are those that interact directly with hardware actuators because we want to block the actuator commands during simulation, however, any procedure can be simulated in order to skip details. All of this requires runtime simulation, which may involve models of various levels of fidelity. The simulation branches may also include choice points and probabilities so that high probability variant outcomes may be simulated. Application processes in planning and execution mode communicate via rules, called Situated Control Rules (SCRs) [Drummond 1989], that provide *choice point advice* about what to do at the choice points embedded in the code.

The **Executive level** is a single process that manages transitions between planning and execution and manages databases that enable inter-process communication and synchronization. This involves sending messages to the supervisor processes. The executive also handles requests for information from the application level.

Three levels but not 3T: Although Propel has three levels, it is significantly different than the 3T system [Bonasso, et., al., 1997]. In 3T, each level encodes domain-specific activities, but in Propel all of the domain-specific activities are encoded only at the application level. Hierarchy within the application level is accomplished through standard C++ programming methodology. Propel's other two layers (Supervisor and Executive) are meta-processes that monitor and manipulate the application level.

Example

Figure 2 shows the application level source code for an example that we will use throughout the paper. The example includes two parallel processes: *Camera* and *Wheels*. The wheels iteratively move to a target while the Camera takes pictures until Camera detects that the wheels have stopped moving, at which time it takes a close-up picture. When the Wheels process arrives at the target, it adds the fact "Not Moving" to the database, which signals to the waiting Camera process that it can proceed to take the close-up. Only the wheels portion is shown due to limited space. Line 58 shows use of the _PLANNING_ flag to determine if a command should be simulated.

Search Space

To recover from software failures, Propel searches through a space of program variations defined choice embedded in the code. The application-level interface to the search engine includes choice points, fail statements, heuristics and meta methods. The choices identify alternative subroutine calls and assignment statements.

Figure 3 shows the Process tree, where each node is a Unix Process. The tree illustrates the three process levels shown in Figure 1. The root node in Figure 3 is the Executive process, the Supervisor processes are the second

```
 1 // Filename: rover.cpp
 2 #include "propel.h"
  ------------------------------------------------
 3 void initializePropel ()
 4 {declare_Process(Wheels);
 5  declare_Process(Camera);
 6  declare_Goal(Move_1, "Move ?d");
 7  declare_Goal(Move_2, "Move ?d");
11  declare_Heuristic(preferFast);
13  declare_Heuristic(preferShort);
16  declare_TP("Not_Moving [1 30]");}
  ------------------------------------------------
17  void Wheels ()
18   {start_Task([5 10] Wheels {1 80} [6 100]);
19    gotoLoc(getTarget());
20    add_Fact("Not Moving");
21    execute_TP  ("Not_Moving", 1, 30);
22    if (!wait_Task ("[0 20] waitForCamera {0 10}
          constraint after Camera End [0 5]"))
23      fail ("Wheels Timeout");}
  ------------------------------------------------
24 void gotoLoc(Location* loc)
25  {start_Task(gotoLoc {0 30});
26   int d, directions[] = {N, S, E, W};
27   int step = 0;
28   _Var(step);
29   while (!atLocation(loc))
30     {d = (int) choose_item(directions, 4,
                "preferShort(?loc)", _Var(loc))
31      choose_task("Move ? :preferFast()", _Var(d));
32      require(step++ < MAX_STEPS);}}
  ------------------------------------------------
48  void Move_1 (void* direction)
49   {Direction dir = *(int*)direction;
50    if (dir == E) agent = NULL; //Fault Injection
51    int x = agent->x;
52    int y = agent->y;
53    switch(dir)
54      {case N: y++; break;
55       case S: y--; break;
56       case E: x++; break;
57       case W: x--; break;}
58    if (_PLANNING_) Simulate_Command("GoTo", x, y);
59    else Execute_Command("GoTo", x, y);}
```

Figure 2: **Application-level source code** for the Wheels process with embedded library calls (in bold text) to search and constraint engines. (The parallel Camera process is not shown due to space limitations)

level, and Application level processes are everything below the second level. Subtrees below the Supervisor processes are search trees, where nodes correspond to application level processes and arcs represent branches at the embedded choice points. The application level processes execute local heuristics to sort the choices using information available at the application level. Global heuristics are used by the SupervisorFailure method for backtracking.

Choice points. Choice points identify alternative operations and resources. Lines 30 and 31 show examples. There are three types of choice points: *choose_item* is a non-deterministic assignment statement, *choose_task* is a non-deterministic subroutine call, and *choose_fact* is a non-deterministic database query. They are non-deterministic because executing them will have different outcomes based on heuristics and planner results.

When a choice point is executed by an application level process, that process calls fork() to create a child process that continues with the selected choice. The parent process remains suspended until backtracking occurs, and then the supervisor may wake up the parent to generate a new choice (fork a new child). Heuristics and other user-specified methods are used for search control.

Figure 3 shows the search space for our example. The nodes at the application level represent computational continuations resulting from forks at choice points Each choice point defines a set of disjunctive branches in the search tree. One branch is created for each choice that is tried. Only choices that are actually tried cause new processes to be created.

`void* choose_item()` - This statement will choose an element from a list of integers or pointers and functions as a non-deterministic assignment statement. The choices are sorted by the specified heuristic function. The <var> are passed into the heuristic. The return value is type void* and points to the object pointer selected from the list.

The example on line 30 says: choose a direction from the array containing all directions (N,S,E,W are C "enums" that map to integers). The heuristic "preferShortest" is used to sort the choices in order of minimum "manhattan distance", and the heuristic takes a target location as input. Heuristics are declared at initialization (lines 10-13).

`choose_task()` - This statement will choose a subroutine that matches the given goal pattern, and functions as a non-deterministic subroutine call. It calls one of several methods that all achieve the same goal pattern. The example shown on line 30 will choose a task that matches the pattern "move ?" where ? is a variable bound to the move direction. This matches the goal declarations from lines 6 and 7, which says that either Move_1 or Move_2 could be called to achieve this goal.

The "?" is a place holder for the goal pattern variables There must be one goal pattern variable provided for each ? in the pattern. This is similar to the way *`printf("%d %d", i, n)`* requires a var for each %. The goal pattern variables are passed by reference so execution of a task that matches the pattern can change the value of the vars. The heuristic is named on the right of the colon, so the example in line 31 says use heuristic "preferFastest".

`choose_fact()-` This statement will choose one fact from the database that matches the given fact pattern, and functions as a non-deterministic database query. It queries the database for facts that match query pattern and returns one of the matching facts. The Executive maintains a

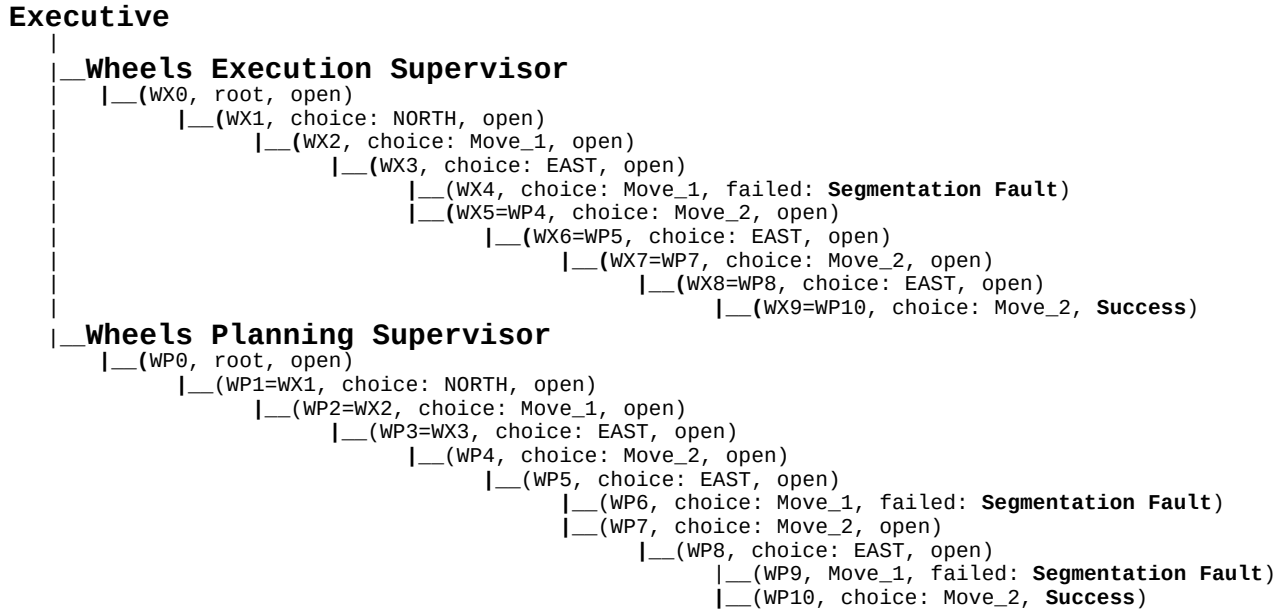

Figure 3: Process Tree and Search Space. The levels of the tree correspond to three levels shown in figure 1. Each node is a process with unique PID. The top level process is the Executive, Level 2 contains supervisors that monitor and manipulate the application-level processes. All nodes below the Supervisors are application-level processes running code defined in figure 2. Each subtree below a supervisor (starting with "root" nodes WX0 and WP0) is a search space with branches that are disjunctive choices.

database that can be used for interprocess communication between concurrent application processes (such as Wheels and Camera). The choose_fact() statement chooses bindings via unification with the DB.

Fail and Require statements. Application level code may contain fail and require statements which cause search node failures and may trigger backtracking. These statements provide runtime verification capabilities by detecting when requirements are violated. Lines 23, and 32 show examples of *fail* and *require* statements.

The **`fail()`** statement causes the current process to inform its supervisor process that it failed, and then the process is suspended. The supervisorFailure() and executiveFailure() message handlers may be designed to handle the failure by transitioning from execution to planning or by backtracking within planning space. The example in line 23 triggers failure if the wait statement in line 22 timesout.

The **`require()`** statement is similar to the C++ Assert statement except that Assert requires user intervention. `require`(condition) will test the condition, calls "fail" if the condition is false. The condition represents a runtime verification requirement. The example in line 32 triggers failure when the variable step exceeds MAX_STEPS.

Other failure types include exhausted choice points, unhandled exceptions, and temporal constraint violations. Any statement may trigger an implicit **fail** statement because any subroutine call has the potential to cause a segmentation fault or floating point exception. Any choose statement can trigger a failure when all choices have been exhausted (or no choices exist). Wait statements trigger a failure if they time out. When any of these failures occurs, the search node marked "failed", the process is suspended, and the supervisor is notified.

Failure Example: We've injected a software failure into our example to show how it is handled. In our example, an unhandled exception occurs because one of the two move operation (`Move_1`) dereferences a null pointer when it tries to move East. Line 50 sets the variable "*agent*" to NULL when the direction is East. This causes the injected failure when the pointer is dereferenced on line 51. The segmentation fault is trapped by propel, and treated as if a fail statement had been executed by the failing process. See the online version of this paper for further explanation of how this failure is handled (Levinson 2005).

Search Control

Propel includes several methods to control search. The first and most important is the *sparse search space*. The search space is sparse because it only has branches at choice point locations. Deterministic subroutine calls like `gotoLoc()` (line 24) are not represented in the search space. Most the application-level code can be deterministic with search triggered only at explicit choice point locations.

Situated Control Rules [Drummond 1989; Drummond et. al, 1993, Levinson 1995] are the second most important search control method. The planner and controllers communicate by exchanging condition-action rules called Situated Control Rules (SCRs). SCRs provide

```
Situated Control Rule:
IF <condition> Then <action>

<condition> = (<StackFrame>+)
<StackFrame> = (Frame: <subroutineEntryPoint>,
                PC <programCounter> <var>*)
<subroutineEntryPoint> =
"subroutineName:lineNumber"
<programCounter> = "filename:lineNumber"
<var> = (Var: <varName> <varAddress> <varValue>)
<action> = (<choice> <mode> <status>)
<mode> = Planning | Execution
<status> = success| failure | open

RULE WX4
IF ((Frame: gotoLoc:rover.cpp:24, PC rover.cpp:31
      (Var: step 0xffbec9d0 0003)
      (Var: d 0xffbec9d4 0003))
    (Frame: Wheels:rover.cpp:18, PC rover.cpp:18))
THEN Move_1 Execution failure

RULE WP10
IF ((Frame: gotoLoc:rover.cpp:24, PC rover.cpp:31
      (Var: step 0xffbec9d0 0005)
      (Var: d 0xffbec9d4 0003))
    (Frame: Wheels:rover.cpp:18, PC rover.cpp:18))
THEN Move_2 Planning success
```

Figure 4: Situated Control Rule (SCR). Each branch in the search space is described by an SCR. The <condition> is the control stack capturing the computational state when the choice is made, and <action> is the choice.

choice point advice that describes preferences at choice pointes. They describe the context and outcome of prior choices made during planning and execution. These rules are the key to context sharing between planning and execution. Figure 4 shows the grammar and examples of SCRs.

One SCR is defined for each branch in the search tree shown in Figure 3. The name of the SCR is the name of the search node in that tree. The condition (left-hand side) of the rule is the control stack when branch occurred, and the action (right-hand side) is the choice associated with that branch. An SCR says: "If the rule's *Condition* part matches the current process' control stack, and the choice outcome was not failure, then select choice (continue the process associated with choice).

In Figure 4, the control stack for Rule WX4 describes the execution context at search Node WX4 (shown in Figure 3). Rule WX4 says: The Wheels procedure was entered at line 18 of file rover.cpp (Figure 2) and then it called subroutine gotoLoc at line 24 of file rover.cpp, and the program counter (PC) says it was at line 31 when the branch occurred (at a choice point). The address and value for gotoLoc()'s local variables *step* and *d* are also recorded in the control stack. This rule will apply only when local variable *step* = 3.

This failure occurred because gotoLoc() called Move_1, which caused a segmentation fault (lines 50-51). After replanning, the executing process will look for planner advice about which choice to take. Rule WP10 says that the planner found a successful plan with choice "Move_2" when the program control stack was in the given state. The rule for node WX4 is used to tell the planner that the controller failed inside the task `gotoLoc()`. The planner uses this to avoid simulation of the "Move_1" choice until it has explored other tasks that achieve the same goal.

SCRs are collected when transitioning between planning and execution and vice versa. Collecting SCRs is similar to classical goal regression and form a partial policy. When a search success or failure occurs, SCRs are collected for the path from the given node to the root node. For example, when the first Wheels excution failure occurs at node WX4 in the tree above, SCRs are collected that describe the control stack, choice, and outcome for each node between WX4 and the root WX0.

The rules are passed from the controller to the planner so that the planner understands what choices the controller took. When the planner finds a workaround, similar rules will be collected from the planner's search tree, and used to tell the controller which choices led to a successful plan. When an application level process executes a choice point, SCRs are combined with heuristics to sort the choices.

Heuristic functions: Users can define *local heuristics* as preference functions that are called to sort choices locally at choice points. The heuristcs can use "less-than" predicates and the built-in function "SortChoices" to reorder the choices at the choice point. For example the following heuristic sorts choices into increasing values. Users can also define *global heuristics* by modifying the search function, **`BestNode`**(), which may control search using global information not available at the application level. BestNode() is called by the supervisors to determine which application level node (process) should be explored (continued) next. Computational state information about application-level processes and the global database can also be used to implement different search strategies.

The default implementation of BestNode is similar to Reaction-First Search (RFS) [Drummond et. al., 1993]. The search is biased to first explore the controller's default behaviors (reflexes defined by heuristics) to see if the controller's "reactions" will work in the current situation. When the default reaction is inappropriate, the planner generates advice rules to override the default reactions.

Supervisor and Executive Interface

The application specific interface to meta processes includes the methods described below. We describe the default behaviors for these methods but users can write their own versions to customize the coordination of planning and execution.

The **SupervisorFailure**() method is called when a Supervisor receives a *SupervisorFailure* message from the application level. In planning mode it triggers backtracking, otherwise it suspends execution and then informs the Executive by sending an *ExecutiveFailure* message. The **SupervisorSuccess**() method is called when the Supervisor receives a *SupervisorSuccess* message from the application level. It typically informs the Executive by sending an *ExecutiveSuccess* message.

The **ExecutiveStartup**() method is the Executive level startup handler called at startup time to initialize concurrent processes and propel structures, including the initial temporal constraint network. This method may start execution before planning or vice versa, or it may run them concurrently, depending on application.

This **ExecutiveFailure**() method is called by the Executive when it receives an *ExecutiveFailure* message from a Supervisor. The Executive knows whether it was a planning failure or an execution failure. In planning mode, this event indicates an exhausted search space.

The **ExecutiveSuccess**() method is called when the Executive receives an *ExecutiveSuccess* message from a Supervisor. This happens when application level process has an empty control stack during execution, or in planning mode when the subroutine which called the failed subroutine is popped off the stack.

These meta-level handlers allow users to specify different *executive strategies* including: a) predictive detection which simulates a program prior to execution, b) reactive detection where you only call the planner after an execution failure occurs, c) anytime planning which stops planning at anytime and collects SCRs for the partial plan, d) batch planning which plans until complete solution is found, and e) incremental replanning which only fixes the current execution failure before returning to execution. Another executive strategy could have the planner generate SCRs for failure contingencies to "robustify" the SCR policy.

Search Process Walkthrough

To illustrate how this works together we will walk through the steps that produced the results shown in Figure 3. To save space, only the Wheels process is described here.

First, the initialization routine shown in Figure 2 is executed. This causes the executive and supervisors to be created along with application-level root nodes WX0, WP0. The root nodes are suspended after creation and the `executiveStartup()` method is executed, which in our case tells the Supervisors to start executing the Wheels process by activating root nodes WX0.

The Wheels execution proceeds through the choice points in the gotoLoc() routine. A branch in the tree is created for each choice point executed. Wheels proceed and generate the nodes WX1, WX2, WX3, and WX4. The node WX4 represents a process that threw a segmentation fault when it executed Move_1 in the East direction. When WX4 calls `fail`(), the Wheels Execution Supervisor is informed, and calls its `SupervisorFailure()` method which may do different things depending on whether it is a planning or execution failure. In this case the Supervisor informs the Executive of the failure and passes the SCR's that describe the failure context.

The Executive's `executiveFailure()` method is called and the Executive informs the Wheels Planning Supervisor to start executing the wheels procedure (in planning mode), and it passes the execution SCRs to the planning Supervisor where they are used to guide the search process down the same path as execution. This guidance can be seen in nodes WP1-WP3, where the nodes are labeled with the name of the execution SCR that was used by the planner. For example WP1=WX1 means that the choice taken by the planner at node WP1 is based on the SCR that defines the branch for execution process WX1. This rule will only apply the first time line 31 is executed because the step variable = 1.

The planner follows the execution SCRs through node WP3 (notice that WP1-WP3 have equals signs). WP4 does not follow the execution choice because that is the choice that led to failure in WX4. This rule, which identifies a choice that led to failure, is a *rule of avoidance*, and treated differently than other rules. SCRs that identify failure choices will cause that choice to be preferred last. It will be chosen only after all other choices failed. The planner delays the execution choice (**`Move_1`**) to the end of the search space since it is known to have failed. The planner chooses the next option, which is **`Move_2`** (node WP4), which does not fail when moving east. The remaining planner nodes (WP4-WP10) are not guided by SCRs because execution never got that far. The planner chooses the failing **`Move_1`** two more times in simulation (nodes WP6 and WP9) before reaching success (empty control stack) in at WP10.

The successful application process WP10 then informs the Wheels Planning Supervisor about the node success and the Supervisor informs the Executive, which executes its `ExecSuccess()` method, and collects SCRs for the path from node WP10 to node WP1. The Executive then informs the Wheels Execution Supervisor to continue execution using the planner's SCRs as choice point advice.

The Wheels Execution Supervisor resumes execution by expanding node WX3 as determined by `BestNode()`. Since WX4 failed, its parent WX3 is the chronological backtracking choice, but it has been suspended since it spawned child WX4. After being reactivated by the Supervisor, Node WX3 generates a new child WX5 based on the SCR from the planner's WP4 node. This instructs the application code to choose `Move_2` instead of `Move_1`. The remainder of the Execution processes (nodes WX5 - WX9) are guided by the planner's SCRs as shown by the equals signs next to the node names.

Temporal Constraint Engine

Propel uses a Simple Temporal Network (STN) [Dechter et al., 1991] to monitor and control C++ program execution based on a declarative temporal model. As the propel application executes, progress is shadowed by the STN. The STN can be partially declared before the C++ applications are started but the network will also be dynamically generated as the C++ code executes.

As execution proceeds, timepoint values are constrained by actual execution times. When a subroutine is called, the STN is checked to see if it is ahead or behind

schedule. If it enters the subroutine early, then it waits. If it is late, then fail() is called and the Supervisor is notified. Static STN statements declare "background" parts of the temporal network that are defined before any tasks start execution, and dynamic statements extend the network dynamically and conditionally during task execution. The following declarative statements can be embedded in C++ to modify the temporal constraint network.

`start_Task()` - Lines 18 and 25 show examples of the start_Task statement, which declares temporal constraints on a C++ function. It creates two timepoints in the temporal network. One timepoint represents the start time of the function and another represents its end. The Task_Start on line 18 declares that Wheels has the following temporal constraints: Start Time in range[5 10], End Time in range [6 100], duration range {1 80}.

`wait_Task()` – wait for a fact to be added to the database, or for temporal constraints to be satisfied (w/ timeout). Lines 22 and 23 show how the wait_Task statement and the `fail` statement can be combined. Line 22 says that the program should start waiting between time 0 and 20 and wait for a maximum of 20 seconds and wait for between 0 and 5 seconds after the Camera program ends. If the maximum duration of 20 seconds is reached before the camera program ends, then `fail`() is called.

`declare_TP()` adds a new Timepoint to the temporal network when it is executed within a C++ function. Other processes may share constraints with TP. Line 16 illustrates the `declare_TP` statement. When this line is executed, a time point is created with lower bound of 1 and an upper bound of 30, so the timepoint must be executed sometime between time 1 and time 30. Other processes can establish constraints to this node using by referring to its label, which is "Not_Moving" in this example. The new timepoint is not actually executed (collapsed to a singleton) until either start_task() or execute_TP() is executed which refers to the same timepoint label.

`execute_TP()` is the same as declare_TP except it also executes the timepoint. This means that the value of the time point is collapsed to a singleton (the current time) and that time is propagated through the temporal network. Line 21 shows an example of execute_TP().

Related Work

Propel 1 [Levinson, 1995; Levinson 1994]. This paper presents Propel 2, which is very different from Propel 1 because it uses compiled C++ as its action representation. In order to accommodate the fact that the application code is compiled, the Propel 2 architecture shown Figure 1 differs significantly from Propel 1's architecture.

Propel 1 had only one planner and one controller, which interpreted the same LISP action representation using their own instruction fetch-execute cycles. In Propel 2, multiple planners and controllers start and stop compiled application code and they don't have instruction fetch-execute cycles. Propel 2 extends Propel 1 by adding temporal constraints. It also extends SCRs to represent the state of compiled C++ code and to include rules of avoidance. Since Propel 2 enables planning to be embedded in C++, it is better suited for use in deployed autonomy applications.

ERE [Drummond, et. al, 1993]- Propel is a direct extension of ERE (the Entropy Reduction Engine). Propel incorporates several ERE features for integrated planning and execution including Reaction First Search and SCRs. Since ERE was never used to model software actions, a key difference is that Propel uses a C++ action representation compared to ERE's STRIPS-like action representation.

IDEA [Muscettola et. al, 2000] The IDEA system is a unified planning and control system like Propel. However, IDEA executes the *planner's* language while Propel plans with the *controller's* language. IDEA's controller executes plans by interpreting the planner's *declarative* language. IDEA models software as black boxes and does not distinguish between a hardware or software black box. It can detect unexpected (software) inputs, outputs, and timing, but has a minimal model of the logic and computational state details relating the inputs to outputs.

KIRK/RMPL – [Kim, et. Al, 2001] William's KIRK/RMPL system also provides a unified approach to planning and execution. It differs from Propel because it compiles procedural constructs into a declarative model which is then interpreted by during execution. KIRK is similar to IDEA this way, but differs from IDEA by using an explicit (declarative) model of control behavior. RMPL can represent control flow constructs such as loops and conditionals, which are compiled into a declarative model used for planning and then interpreted during execution.

Future Work

Propel 2 is currently in the working prototype stage. We have identified many open research issues including:

Backtracking issues such as using model-based diagnosis to provide dependency directed backtracking. We also must address issues such as deciding which concurrent processes must be planned together, and simulation with metric time (backtracking and warping forward).

Executive Strategies for managing transitions between planning and execution. This includes proactive planning, concurrent, and interleaved planning, anytime planning, and planning after a failure occurs. This involves definition of the planner termination test which decides when the planner has "gotten around" the current failure so that execution may continue.

Software Sensors and Actuators. We currently insert macros to instrument the code by hand. Future work may use a separate preprocessing phase to automatically instrument the code, and also OS level instrumentation of computational state. We'd like to use OS-level actuators that may provide lightweight alternatives to fork(). We also need a better way to capture the control stack information used by SCRs.

Runtime Simulators are needed so the application code can run in _PLANNING_ mode (see Line 58). The simulators are needed only for physical actions and may provide different levels of abstraction and/or fidelity. Users can plug in application-specific simulators or use Propel's built-in database to keep track of simulated or executed state properties. Planning and execution have their own copies of the database.

Performance - The search nodes are currently implemented as computational continuations (created by the UNIX fork() command). Future work will involve using lower-overhead alternatives to fork(). Also could use branch and bound to limit the number of processes that remain open for backtracking.

Evaluation Plan

We will perform experiments to test our hypothesis that unified planning and execution with a procedural representation can significantly increase failure recovery scope and decrease cost. We will inject software failures into complex software and measure the coverage of existing recovery systems compared to our approach. We will measure the costs for human vs. autonomous recovery, and performance costs of the new methods.

Conclusion

Propel is a unified planning and execution system that uses a procedural representation. This is different from IDEA, which exclusively uses a declarative action representation.

Since most software is not written as a declarative model it tends to be outside the scope of a planner's reasoning. PROPEL was designed to increase the scope of the planner's model to include software in order to address the problem of software failure detection and recovery.

Propel was designed to close the gap between the declarative action model used by a planner and the procedural languages used to develop real-world software. The representation is intended to be expressive enough to be used in system software including the planner and executive software. Motivation for using a procedural representation includes the following goals:

- Include all software within the planner's model in order to increase the scope of failure recovery to include infrastructure software failures.
- Represent complex procedures including loops, conditionals, local variables, and multiprocessing.
- Reduce the need to develop and maintain different models for the planner and execution system.
- Reduce risk of loss of information in translation between execution and planning (and vice versa).

Propel is both an architecture and a language. The architecture provides integrated planning and execution modules that monitor and manipulate application-level processes written in the Propel language. The language is a library of methods for embedding search and temporal constraint information into C++, thus creating a "superset" of C++ like TDL. This library provides an interface from the Propel application code to the supervisory meta-processes (the planning and execution modules), which monitor the application to provide failure detection and recovery.

The language provides an action representation that captures control constructs and can also be projected by a search-based planner. The planner can provide a useful partial plan even when it is interrupted after an arbitrary amount of computation. The planner and the controller share identical data structures and algorithms for interpreting a shared representation of control actions.

The other unified planning and execution systems emphasize recovery from hardware failures more than software failures.

References

Bonasso, R, Firby R., Gat, E., Kortenkamp, D., Miller, D., and Slack, M. Experiences with an Architecture for Intelligent, Reactive Agents, in *Journal of Experimental and Theoretical Artificial Intelligence* , January, 1997.

Dechter, R, Meiri, I. and Pearl, J. 1991. Temporal Constraint Networks. *Artificial Intelligence*, 49:61-95.

Drummond. M. Situated Control Rules. 1989. *Proceedings of Knowledge Representation 1999* (KR'89).

Drummond, M., Bresina, J., Swanson, K., Levinson, R. 1993. Reaction-First Search: Incremental Planning with Guaranteed Performance Improvement. *Proceedings. of IJCAI-93*. Chambrey, France.

Kim P., Williams B., Abramson M., 2001. Executing Reactive, Model-based Programs through Graph-based Temporal Planning. IJCAI '01. AAAI, Menlo Park, CA.

Levinson, R. 1995. A General Programming Language for Unified Planning and Control. *Artificial Intelligence*, Vol. 76. See http://www.brainaid.com/papers/propel_aij95.pdf.

Levinson R. 2005. Unified Planning and Execution for Autonomous Software Repair (10-pg version of this paper) http://www.brainaid.com/papers/propel_ijcai05.pdf.

Muscettola, N., Dorais, G., Fry, C., Levinson, R., Plaunt, C. 2000. A Unified Approach to Model-Based Planning and Execution. *Proc. of IAS '2000*. Venice, Italy.