Exploring Alternative Software Architecture Designs: A Planning Perspective

J. Andrés Díaz-Pace, *Software Engineering Institute, Carnegie Mellon University*

Marcelo R. Campo, *UNICEN University*

The DesignBots framework supports architects in searching for design alternatives by capturing quality-attribute design concepts into a hierarchical, mixed-initiative planning model.

Software architecture designs give us blueprints to build systems, enabling key early decisions that can help us achieve a system's functional and quality-attribute requirements.[1] Architectural decisions have far-reaching effects on development in terms of quality, time, and cost. Architects apply technical knowledge and experience to guide their decision making, choosing among multiple design solutions to find a reasonable balance of quality attributes such as performance, modifiability, or security. This is complex and time consuming because qualities can conflict and lead to trade-offs. A trade-off means that the improvement of one quality comes at the cost of degrading another—for example, modifiability versus performance.

Since the mid-2000s, abstractions and techniques for architectural design have been steadily improving. Of particular interest are the pioneering efforts of the Software Engineering Institute, which has developed a "theory for predictable architecture design" to manage the relationships between quality-attribute issues and architectural decisions.[2] According to this theory, qualities don't arise spontaneously from an architecture; rather, architects *plan for* qualities by articulating predefined architectural mechanisms.[1]

Architects usually start with an initial architectural solution, and then progressively consider improvements regarding a few quality-attribute drivers. As decision making proceeds, architects explore, evaluate, and compose architectural transformations. Approaches such as *predictable architecture design* (PAD) can help them explore alternatives systematically. Moreover, architects can benefit from tools that intelligently navigate the design space. Nonetheless, PAD concepts don't support design's explorative aspects per se. To address this problem, we treat exploration as a type of search in which architectural knowledge prunes options and directs the architect toward "good-enough" solutions. Along this line, AI planning is a suitable technique to (semi)automate that search. Essentially, the quality-attribute drivers would be the goals that appropriate architectural transformations (that is, basic actions operating on components and connectors) must satisfy. Our objective is to develop a design assistant that lets architects focus on the key decisions for shaping the architecture by delegating to a planning engine the routine search work derived from those decisions.

The DesignBots framework is a prototype for such an assistant. It provides a multiagent infrastructure that maps PAD concepts to a hierarchical task network (HTN) planning model.[3] HTN planning represents plans as task hierarchies that can be gradually refined into subplans. In the context of PAD, HTN planning helps identify tactical solutions for quality attributes and then provide separately the details of their materialization via architectural

1541-1672/08/$25.00 © 2008 IEEE
Published by the IEEE Computer Society

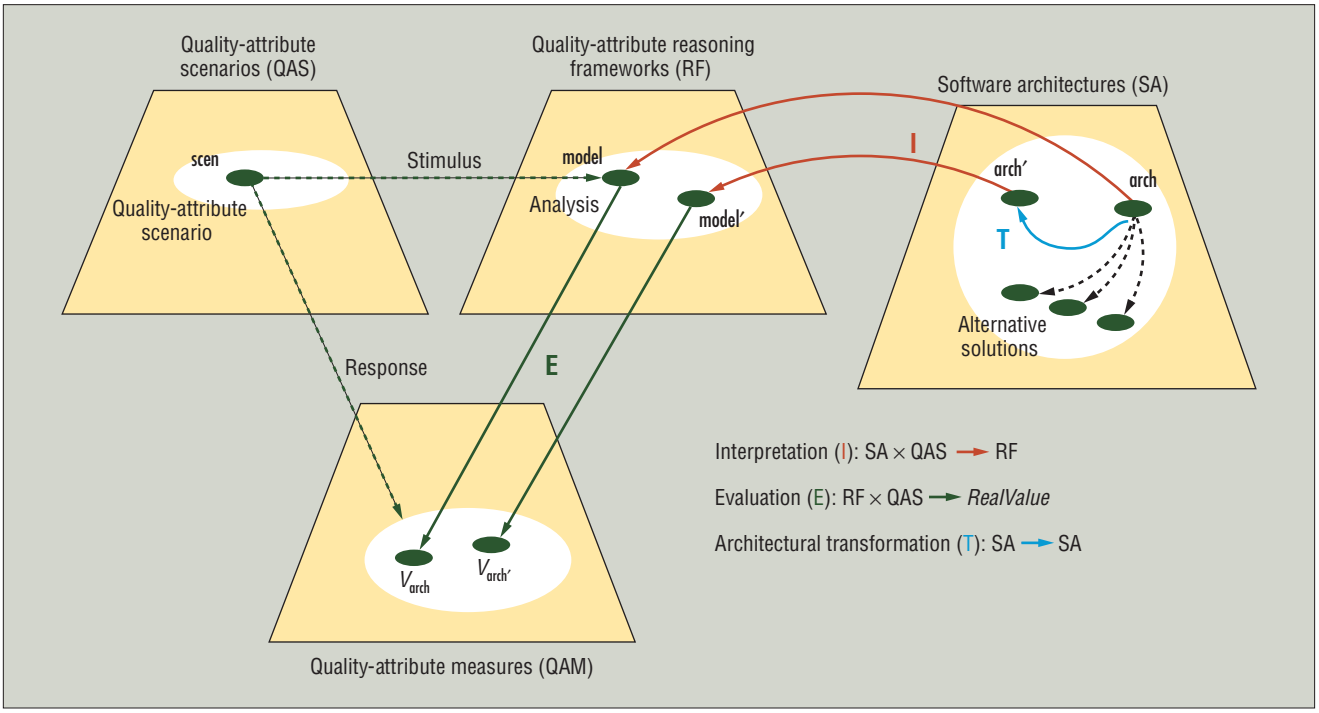


Figure 1. Architectural theory for reasoning about quality-attribute design.[2] The design space is divided into four planes: the quality-attribute scenario plane, the quality-attribute measure plane, the software architecture plane, and the quality-attribute reasoning framework plane. The first two planes represent the problem space, the third represents the solution space, and the fourth captures the quality-attribute analysis that connects the problem and solution spaces.

patterns. This enables the (semi)automated exploration of alternatives for an input architecture and a set of quality-attribute scenarios. To study this approach's feasibility, we equipped DesignBots with architectural knowledge for modifiability and performance. Then, we empirically compared the prototype outputs against human designs from case studies. The results regarding mappings and search capabilities show potential for planning techniques to support architectural tools based on quality-driven constructive principles.

Architectural design as a planning problem

One premise of the software architecture community is that quality attributes can be realized by applying specific architectural patterns.[1,4] In this context, a central question is how to move from a set of quality-attribute requirements to an architectural structure that satisfies those requirements. For instance, let's suppose a modifiability scenario stating that component A should incorporate new features with reduced change impact. A possible architectural strategy for the scenario is to insert an intermediary between A and its interacting components so that the intermediary can break A's dependencies on other parts of the system. Depending on the dependency type, the architect would use various patterns to flesh out a solution—namely, a repository, a naming server, a publisher-subscriber, layers, and so on.[4] Several researchers have investigated providing systematic "reasoning threads" for this kind of design (see the sidebar "Related Work in Automated Design Support"). Currently, one of the most representative approaches is PAD,[2] which establishes guidelines for building architectures with predictable quality-attribute properties based on three essential concepts: quality-attribute scenarios, reasoning frameworks, and architectural tactics. Figure 1 depicts the links among these concepts, given a particular quality attribute.

The PAD process is as follows. We assume an initial architecture **arch** as a formal specification of components, allocation of responsibilities to components, and connectors among components. We express a quality-attribute requirement as a *scenario*, which is a textual description of a use case for the system. A scenario **scen** defines an envelope of allowed quality-attribute measures. An interpretation function **I(arch, scen)** analyzes the architecture through a reasoning framework and instantiates a **model** instance. This reasoning framework in turn applies an evaluation function **E(model, scen)** to determine the quality-attribute value (or response) that such an architecture will achieve for a scenario stimulus. If the evaluated value is inside the region defined by the scenario, the architecture will be "good enough" to satisfy the scenario. If not, a *tactic* changes the architecture using a transformation function **T(arch)** so that a parameter of the reasoning framework moves in a known direction and its (re)evaluation falls into the desired region. Tactics capture the logic behind architectural patterns to tackle "classes of quality-attribute problems" (for example, rippling of changes for modifiability and bottlenecks for performance).[1] Furthermore, a tactic gives directives to control reasoning-framework parameters. In the example just given, the tactic of "inserting an intermediary" affects the probability of change rippling, which is one of the parameters used by the modifiability framework to compute the total cost of modifications. Tactical directives are made concrete through one or more architectural patterns that can actually transform the architecture.

Related Work in Automated Design Support

Researchers have done considerable work in automated design support, but little in architectural-design approaches using AI. Interesting automated-design-support approaches include rule-based systems,[1] goal-feature graphs,[2] planning for project management,[3–5] multiobjective optimization,[6] and reconfiguration of distributed systems.[7,8]

The NFR (nonfunctional requirements) Framework treats quality attributes as a graph of synergistic or conflicting goals.[2] This approach has codified knowledge about satisfying these goals into methods, which work like their hierarchical task network counterparts, and correlation rules, which deal with general trade-off analysis. However, this framework doesn't provide details regarding architectural structure. A later version of the NFR Framework added a feature-solution graph that connects requirements with architectural fragments. One difference with DesignBots is the lack of (automatable) guidelines for exploring alternatives through the feature-solution graph.

The first experiments with rule-based architectures for design can be traced to the Programmer's Apprentice project at MIT. Unfortunately, much of that work failed because of the underlying design theory's weak support. The Software Engineering Institute developed the ArchE expert system (www.sei.cmu.edu/architecture/arche.html) on the basis of *predictable architecture design* (PAD) concepts to help architects quickly explore design alternatives. The use of rules imposes limitations when specifying complex decision procedures. Open issues for both ArchE and DesignBots include the amount of data generated during search, interaction with the architect, and the management of trade-offs. Currently, research efforts are oriented toward improving assistance using other AI techniques.

John Clarke and his colleagues discuss a view of software engineering as a search framework.[6] According to early results, a well-defined mapping from software concepts to a particular optimization technique is an important requirement. Overall, we still don't know whether optimizations can handle complex design spaces in acceptable time and with a good diversity of solutions.

Other researchers have applied temporal planning to distributed-systems reconfiguration.[7] Although this approach is still in an experimental stage, it concurs with our observations about domain writing and planner scalability. David Garlan, Shan-Weng Cheng, and Bradley Schmerl[8] propose a more flexible reconfiguration approach that makes architectural information explicit at runtime and provides a mapping between architecture and code. This permits detection when system behavior falls outside the acceptable range and modification of component configuration accordingly. Even though this reconfiguration is based on rules, it applies many architectural strategies that designbots use.

Two planning approaches related to project management are RealPlan[4] and Cabma.[5] RealPlan treats resources as separate from causal reasoning, using scheduling to allocate enough resources after selecting actions to reach the goals. This is expected to improve planner efficiency when resources are at work and replanning might be needed. Because the algorithm considers architectural elements as resources, DesignBots could apply it to manage interactions between solutions. Cabma combines HTN planning and case-based reasoning to reuse pieces of project plans, helping users create new projects. Thus, Cabma naturally supports a mixed-initiative modality of interaction. Barbara Dellen and Frank Maurer have also used planning for process management; their tool performs the processes and guides project members to carry out activities.[3] The approach includes tasks, methods, and agents (both human and machine), all of which resemble some of the DesignBots concepts. Because it concerns architectural design, however, the process gives just a general schema of the product design. Additional subprocesses are scheduled as the design proceeds. In contrast, DesignBots has a more elaborated design theory in which the artifacts are more important than the enactment of processes. Thus, the design process in DesignBots doesn't need agents to be able to work. DesignBots (as inherited from PAD) includes criteria to measure the degree of quality-attribute achievement of the solutions. Moreover, the agents in our framework reduce the complexity of designing with quality-attribute knowledge.

References

1. F. Bachmann et al., "Designing Software Architectures to Achieve Quality Attribute Requirements," *IEE Proc. Software*, vol. 152, no. 4, 2005, pp. 153–165.
2. L. Chung, B. Nixon, and E. Yu, "Using Non-Functional Requirements to Systematically Select among Alternatives in Architectural Design (1995)," *Proc. 1st Int'l Workshop Architectures for Software Systems*, 1995, pp. 31–43, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.2252.
3. B. Dellen and F. Maurer, "Integrating Planning and Execution in Software Development Processes," *Proc. 5th Int'l Workshop Enabling Technologies on Infrastructure for Collaborative Enterprises* (WET ICE 96), IEEE CS Press, 2006, pp. 170–176.
4. B. Srivastava, S. Kambhampati, and M. Do, "Planning the Project Management Way: Efficient Planning by Effective Integration of Causal and Resource Reasoning in RealPlan," *Artificial Intelligence*, vol. 131, nos. 1–2, 2001, pp. 73–134.
5. K. Xu and H. Muñoz-Avila, "Cabma: Case-Based Project Management Assistant," *16th Conf. Innovative Applications of Artificial Intelligence* (IAAI 04), AAAI Press, 2004, pp. 931–936.
6. J. Clarke et al., "Reformulating Software Engineering as a Search Problem," *IEE Proc. Software*, Institution of Eng. and Technology, vol. 150, no. 3, 2003, pp. 161–175.
7. N. Arshad, D. Heimbigner, and A. Wolf, "Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems," *IEEE Int'l Conf. Tools with Artificial Intelligence* (ICTAI 03), IEEE CS Press, 2003, pp. 39–46.
8. D. Garlan, S.-W. Cheng, and B. Schmerl, "Increasing System Dependability through Architecture-Based Self-Repair," *Architecting Dependable Systems*, LNCS 2677, R. de Lemos, C. Gacek, and A. Romanovsky, eds., Springer, 2003.

From an operational perspective, architects use tactics and patterns to explore the design space and build architectural solutions. Note, however, that the choice of suitable mechanisms to transform an architecture is implicit in PAD and thus left to search. To provide automated assistance for this search, we argue that the transformational knowledge embedded in tactics and patterns fits well with HTN planning techniques.[3] Assuming tactics and patterns as the planning domain, the two assets re-

quired to complete the mapping of PAD to HTN are the initial architecture as the world state, and this architecture's scenarios as the quality-attribute goals to plan for. Finally, the resulting plans are actually what will generate the architectural alternatives.

The HTN planning problem for PAD is stated formally as the 4-tuple <ArchitectureState, QATaskNetwork, PatternDomain, Plan> with the following elements:

- ArchitectureState is a set of ground atoms (logical predicate names, each followed by a list of arguments that are all bound to constant values). Within the software architectures (SA) plane, these ground atoms describe the components, connectors, and main functions that constitute the architecture.
- QATaskNetwork is a pair <Tasks, O>, where Tasks is a sequence of ground tasks and O is a partial order for them. A task is an atom denoting an activity to be accomplished. In the quality-attribute scenarios (QAS) plane and the quality-attribute reasoning frameworks (RF) plane, QATaskNetwork is a partial order of outstanding goals to be satisfied in the specified order. Tasks capture actual quality-attribute issues coming from the analyses performed via the I(arch, scen) and E(model, scen) functions.
- PatternDomain is a collection of HTN procedures for individual tasks. Architectural tactics and patterns are represented in terms of HTN methods and operators whose instantiation implements the T(arch) function.
- Plan is a sequence of ground tasks that defines a transformation T(arch). When the planning engine generates a successful plan, the plan instance is also a planning state. So, Plan is the partial solution found so far, and the engine has already applied the corresponding transformation to the architecture. This transformation is quality driven in the sense that it follows from goals for a quality attribute, and it's also ruled by a tactic adequate for that attribute. Reevaluating the reasoning framework can quantify the architectural improvement.

A HTN method within PatternDomain has the form <Head, Pre, ApplyWhen, Body>. Head refers to the atom used as the method name (which the planning engine can match against tasks). Pre defines a list of logical preconditions that must be true on ArchitectureState for the method to be applied. Body specifies a task refinement as a new network <Tasks, O>. Within the SA plane, this refinement will capture the design decisions leading to a pattern (or part of it), but it won't make any changes to ArchitectureState. ApplyWhen acts as a filtering condition for the method to check its alignment with a particular tactic.

A HTN operator within PatternDomain has the form <Head, Pre, ListAdd, ListDel>. Head and Pre are similar to the method counterparts, whereas ListAdd and ListDel are lists of logical atoms. An operator changes ArchitectureState by removing the atoms in ListDel and adding the atoms in ListAdd. Within the SA plane, an operator will fill in pattern implementation details (left blank by precedent methods) that lead to concrete architectural modifications.

Design alternatives to improve the initial architecture are expected to emerge from the cooperative work of all the agents (along with the architect).

The basic HTN planning strategy entails a trial-and-error search evaluating multiple possibilities until it finds a transformation that works for the current architecture. Algorithms such as SHOP2 provide efficient implementations of this strategy.[3] Nonetheless, because of the inherent complexity of designing with quality-attribute trade-offs, additional features are necessary to keep the search computationally tractable. According to PAD, we design for an individual quality, as the set of planes in Figure 1 show. This procedure (and thus the set of planes) can be replicated to deal with other qualities of interest, assuming all the parties share the SA plane. If the architecture doesn't meet a scenario, PAD suggests two alternative courses of action:

- the planning engine keeps examining the design space for alternative transformations, or
- when no alternatives exist, the architect "softens" the quality-attribute response of one or more scenarios.

DesignBots tackles these aspects by complementing the HTN planning formulation with multiagent technology and a mixed-initiative modality of interaction.

The DesignBots approach

DesignBots is a multiagent framework that supports planning-based design assistance. The framework divides the architectural knowledge into agents, referred to as *designbots*. Different types of designbots have competencies in different qualities (for example, performance-oriented agents and modifiability-oriented agents). This view captures the usual division of expertise regarding quality-attribute design techniques.[1] The framework also receives two inputs: an initial architecture and a weighted list of quality-attribute scenarios for that architecture. Design alternatives to improve the initial architecture are expected to emerge from the cooperative work of all the agents (along with the architect).

Figure 2 (see p. 70) shows the main DesignBots workflow. During setup, each designbot is configured with a reasoning framework and architectural tactics and patterns as PAD has prescribed. The agents process their respective scenarios to derive goals and then rely on a HTN planning engine to generate architectural transformations. A special agent called a *mediator* combines the designbots' plans into a global transformation. The designbots' goals are prioritized according to the scenarios' relevance for the system to balance the individual plans' effects on the architecture. Once candidate transformations are available, the architect can select any of them and proceed to modify his or her architecture. This exploration continues until the architect achieves a design that satisfies his or her expectations.

Even with PAD, agents can't make certain decisions, simply because the agents cannot have complete knowledge (or enough evidence) to prefer one option over others. Usually, architects are good at solving certain parts of a problem on the basis of their experience (for example, trade-off resolution, preferences for pattern variants, and business considerations), although they're not always able to provide a rule for their decisions. For this reason, DesignBots implements a

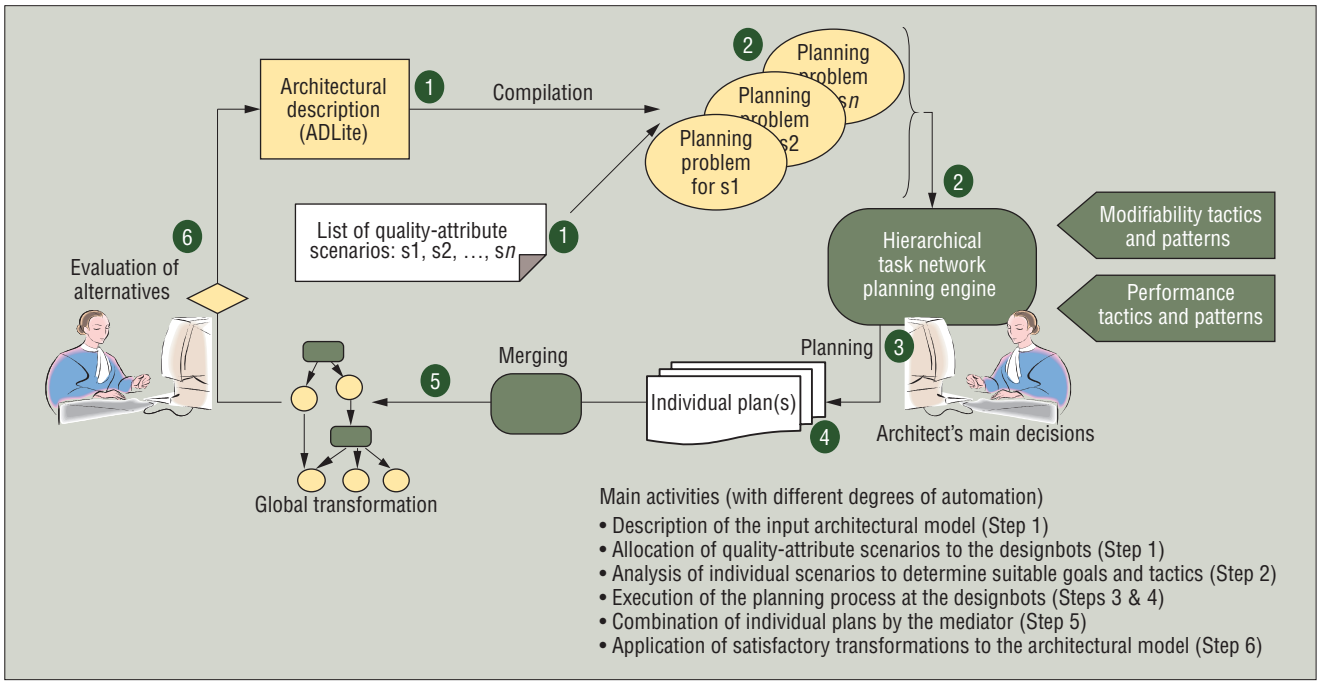


Figure 2. Flow of activities in the DesignBots framework. The architectural description and its quality-attribute scenarios are compiled into planning problems. The hierarchical task network (HTN) planning engine tries to find plans that solve the problems. During the planning process, the engine can call the user (architect) for intervention. The final plans are merged in the form of architectural transformations.

mixed-initiative planning modality that puts the architect in the search loop. As long as the architect makes the principal decisions, the planning engine can consider subsidiary decisions for the plans as well as ensure a correct application of patterns. For instance, if the planning engine detects a problematic component dependency, it would show the architect a list of available patterns for breaking the dependency. Once the architect picks a particular pattern, the engine can continue the search on the basis of that pattern.

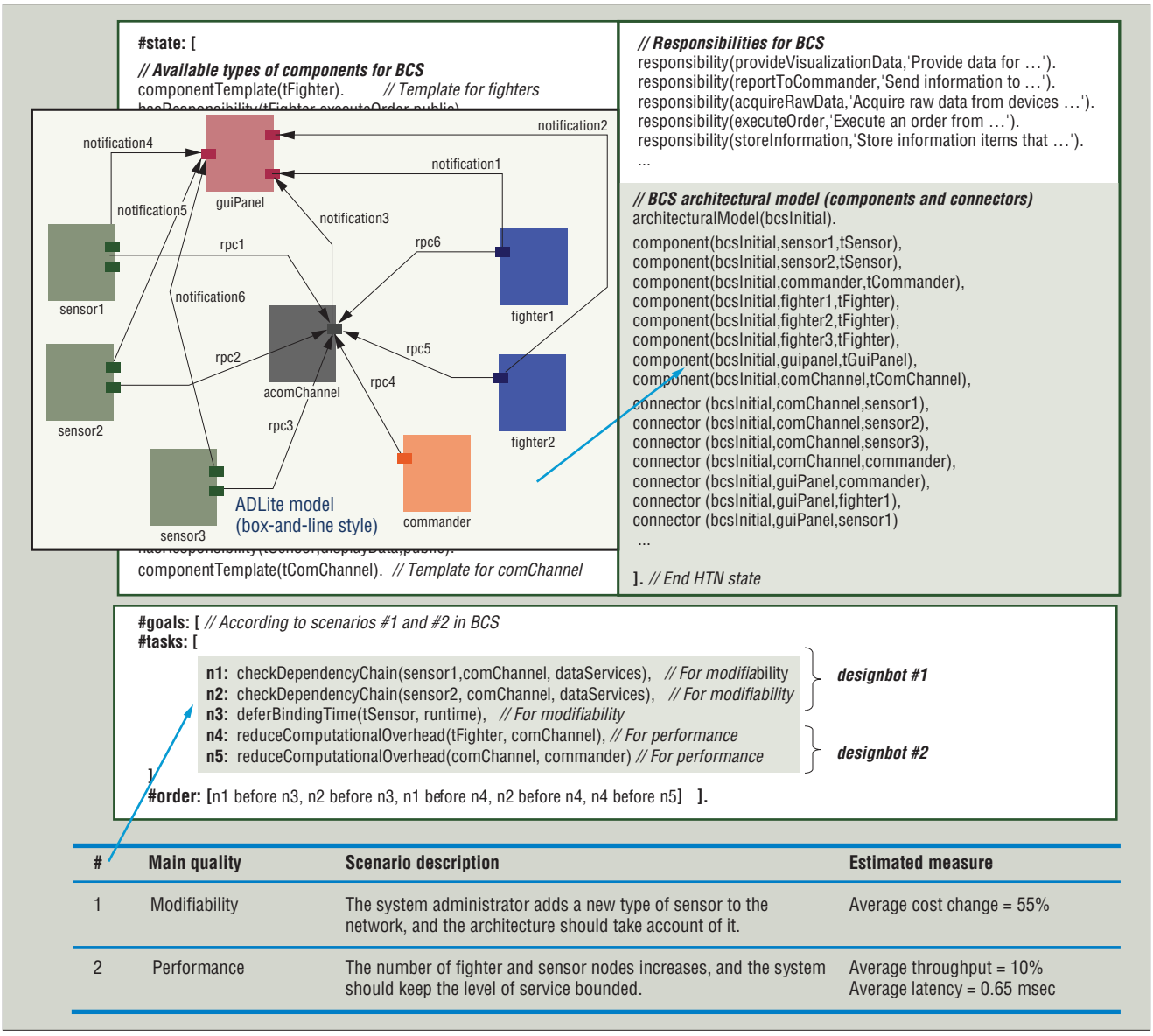
Later, we'll explain the "constructive" aspects of our planning system when exploring modifiability and performance alternatives. A simplified battlefield control system (BCS), adapted from the work of Rick Kazman and his colleagues, illustrates the approach.[5] BCS involves a central commander and a collection of army units (for example, troops, tanks, planes, sensors, and maps). The initial architecture follows a client-server pattern[4] in which the commander acts as the server and the units are its clients, either making requests or updating the commander's database. Internode communication occurs through messages sent via a shared communication channel. All these components display their computations in a graphical interface.

The architectural model and quality-attribute goals

Architectures are commonly represented as graphs of interacting components. This view is supported by architectural notations known as *architectural description languages* (ADLs). For DesignBots, we've defined an ADL called ADLite that gives a basic vocabulary of components, connectors, and responsibilities. The main units of computation are the components (processes, clients, servers, and repositories). Connectors model pathways of interaction between components (procedure calls, events, and access to shared data). Responsibilities capture application-specific functions that the architect assigns to components. In addition, ADLite elements can be annotated with properties. An architectural transformation is meant to change the actual configuration of components, connectors, and responsibilities. A special translator compiles ADLite specifications into world-state facts. Figure 3 shows the BCS architecture in ADLite and the corresponding HTN script. We simplified ADLite's ADL constructs to facilitate the analysis and transformation of architectural models. However, architectures expressed in ADLite can be equivalently specified with other notations such as UML2 or Acme.

A list of scenarios accompanies the input architecture. A quality-attribute scenario captures a textual story of (desired) system usage. Each scenario must involve a single quality and a response level (for example, throughput for performance and cost of changing components for modifiability). Two scenarios elicited for BCS appear at the bottom of Figure 3 (along with initial estimates of responses). The architect determines a ranking for the scenarios and distributes them among the designbots according to target quality. Let's assume Designbot 1 is a specialist in modifiability and Designbot 2 is a specialist in performance, and both are set to BCS Scenarios 1 and 2, respectively. Let's also consider that Scenario 1 is more important for the architecture than Scenario 2.

The designbots employ *architectural scopes* to analyze the scenarios via reasoning frameworks. A scope isolates the area of the architecture affected by a scenario. We added scopes to PAD interpretation and evaluation to narrow the planning engine's world state and to help designbots better derive the goals they're pursuing. Identifying scopes requires the architect's cooperation. He or she must specify the responsibilities implied by the scenario and then execute



#	Main quality	Scenario description	Estimated measure
1	Modifiability	The system administrator adds a new type of sensor to the network, and the architecture should take account of it.	Average cost change = 55%
2	Performance	The number of fighter and sensor nodes increases, and the system should keep the level of service bounded.	Average throughput = 10% Average latency = 0.65 msec

Figure 3. Initial architecture of and quality-attribute scenarios from our example battlefield control system. The graphical model of components and connectors in ADLite (on the left) becomes a set of facts for the world state. The textual scenarios (at the bottom) lead to a network of goals. The world-state facts and the goals are the inputs to the HTN planning algorithm.

algorithms (provided by the framework) to traverse the architecture and create a view (the scope) containing the components that are related to the responsibilities. Once the scope is determined, the reasoning framework runs its quality-attribute analysis as usual. DesignBots implements two reasoning frameworks: a queuing model for evaluating performance and a dependency-chain model for evaluating modifiability. (A reasoning framework is considered here as a black-box library for computing scenario responses.) During analysis, the components, connectors, and responsibilities participating in the scope are annotated with quality-related properties.

On the basis of the scope information, screening rules decide which reasoning-framework parameters should be manipulated to affect the scenario response.[2] Specifically, each designbot applies these rules to establish a list of candidate tactics and a network of goals (that is, tasks) for its scenario. From these tactics, the architect chooses one tactic per scenario as the driving force for further task-network refinements via planning.

For example, in the case of Scenario 1 for modifiability, Designbot 1 calls a dependency-chain reasoning framework to compute the cost of changing the sensors. In short, the interpretation considers a graph with three types of elements: primary nodes, secondary nodes, and the links among them. A primary node is a component whose responsibilities are directly affected by the specific change. A secondary node is a component that interacts with a primary node. A link is a connector through which changes can propagate from primary to secondary nodes. Each node, whether primary or secondary, is characterized by several cost

properties, whereas each link is assigned to a change-propagation probability. A weighted sum over the scope elements gives the total cost as the scenario response. Looking at the response for Scenario 1 (see the table in Figure 3), the architect would judge that a value above 50 percent isn't good. Then, screening rules would point out a problematic dependency chain from the sensors to the commander. To minimize the cost of change, Designbot 1 can suggest the tactic of inserting an intermediary between the sensors and other components. Here, the rationale is that the fewer components reached by the change, the less effort required to support it. So, we'd expect a better modifiability response. Analogously, Designbot 2 suggests the tactic of keeping the communications latency under a specific value to control Scenario 2's performance response. At last, the designbots will infer their goals by relating the quality-attribute analysis results to the components and connectors within their scopes. Actually, when used in conjunction with scenarios, goal-based networks are useful for directing design efforts. Figure 3 shows a task network inferred for the two BCS scenarios. (Because of space limitations, we can't provide the whole analysis process through which the designbots arrive at these goals.) We can generate ordering constraints on the basis of scenario priorities.

Tactics and architectural patterns

The designbots rely on a set of architectural patterns and mechanisms—that is, design elements with different levels of granularity that serve to make tactics concrete.[1,4] An architectural mechanism is fine grained and implements a single tactic, whereas an architectural pattern is coarse grained and usually encapsulates one or more tactics at the same time. When mapped to the HTN domain, architectural patterns and mechanisms can range from basic actions (creation and deletion of components or allocation of responsibilities) to more complex ones (delegation of responsibilities or insertion of a blackboard as intermediary). The bottom of Figure 4 shows scripts for basic mechanisms. We can write more elaborate mechanisms and patterns on top of basic ones until they reach the level of tactical tasks. The top of Figure 4 shows examples of composite scripts. In particular, the thread starts with the tactic of breaking the dependency chain and the high-level task **checkDependencyChain (?primary, ?secondary, ?dependency)**. The first two goals in Figure 3 (for Scenario 1 in BCS) are instances of that schema.

The arrows in Figure 4 indicate how decomposition proceeds. Initially, the task **checkDependencyChain()** verifies whether the architect wants to act on the dependency between two designated components. According to the mixed-initiative modality, user-query tasks permit information gathering at planning time (**@selectOption()**, **@warning()**, and **@getInput()**). The task **@selectOption()** displays a GUI panel and asks the architect for a decision. If the architect enters a positive answer, the next task is to effectively break the dependency with an intermediary. If not, the system sends a warning message to the GUI panel. There are various implementations for the intermediary following this tactic. The first option would replace the actual connector with a new one that confers weak coupling; the second option would insert a new component to bridge the components. If the architect selects the latter option, the tactic is implemented by means of the Forwarder-Receiver pattern.[4] This pattern provides transparent interprocess communication (IPC) within a peer-to-peer interaction model. The top-level method **applyForwarderReceiver()** specifies the arrangement of forwarder and receiver roles as well as their responsibilities. Subsequent methods and operators provide implementation details for the pattern.

> The designbots rely on a set of architectural patterns and mechanisms—that is, design elements with different levels of granularity that serve to make tactics concrete.

Planning for design alternatives with SHOP2

The designbots yield control to the planning engine to get plans for their task networks. This engine uses a SHOP2-like algorithm,[3] generating the steps of a plan in the same order those steps are to be executed. We took SHOP2 mainly because of its performance and substantial expressive power. As in SHOP2, when the planning engine is decomposing a task network, methods can specify subtasks that are just partially ordered. This way, the engine (and the architect) has freedom to decide the tasks to work on, and task precedence is enforced only when necessary. A sketch of our algorithm dbotsSHOP2 appears in Figure 5. This algorithm takes as inputs an initial state **S** (the designbot's scope), a task network **Goals**, the name of a **SelectedTactic**, and a domain **D**. Basically, dbotsSHOP2 is a loop that picks a task **t** in **Goals** that has no predecessors and searches for reductions **<primTask, subtasks>** for that task. The termination conditions are either failure or no more goals. If **Goals** is empty, dbotsSHOP2 returns a plan **outputPlan** and a modified state S_{new} to the requesting designbot. Although SHOP2 was originally designed to avoid backtracking, we equipped dbotsSHOP2 with backtracking points at which designbots can ask the planning engine for alternative-path decomposition if needed. The main backtracking points comprise generation of a task network for a tactic, execution of a task without predecessors within the network, and selection of HTN methods and operators whose preconditions hold in the world state.

A graphical interface controls the planning process. This supports plan construction through dialogues between the architect and the planning engine. Furthermore, the architect can selectively enable or disable backtracking points, undo or redo capabilities, and perform step-by-step execution at configuration time. When the planning engine recognizes a user-query task **primTask**, the system adds the task to **todoGUIList** and the planning engine suspends treating the successors of **primTask**. Through **todoGUIList**, the architect is aware of the decisions pending for the design solution under consideration and can opportunistically answer the required tasks. As long as the planning engine finds tasks ready for execution, it keeps processing the task network.

So far, we've talked about how a designbot pursues one quality-attribute scenario at a time. To account for quality-attribute trade-offs, the framework must handle interactions among goals and their plans from a unified perspective. The mediator comes with two techniques to coordinate designbots' activities.

One technique is to order the goals and solve them linearly. This technique assumes

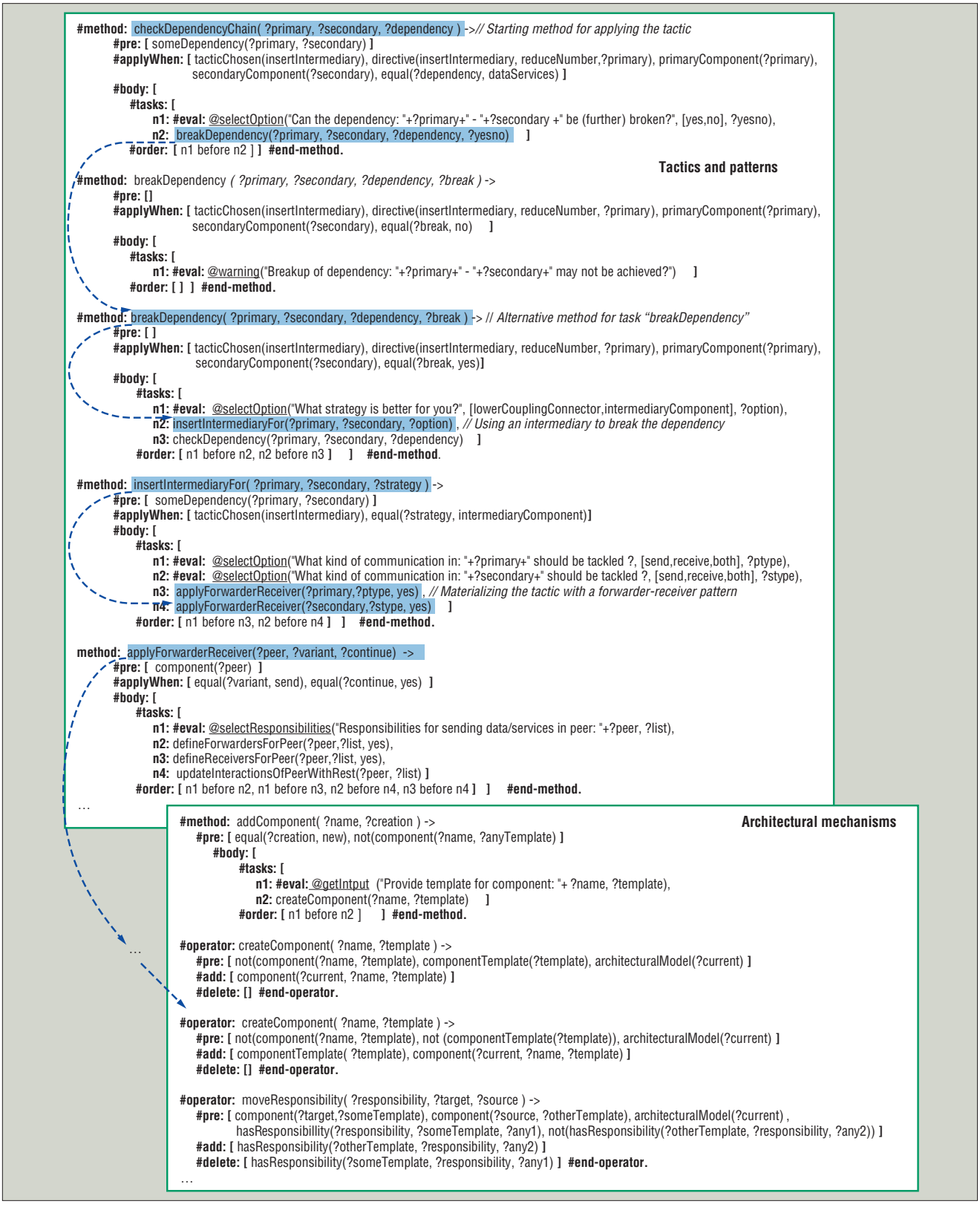


Figure 4. HTN scripts for tactics and architectural patterns. The code at the bottom contains scripts for basic architectural mechanisms, and the code at the top shows composite scripts for tactics and patterns. The arrows illustrate the task decomposition structure.

```
ALGORITHM dbotsSHOP2(S, D, SelectedTactic, Goals): < Plan , S_new > {
  outputPlan ← Ø  pendingTasks ← Ø
  tasksAvailable ← {t ∈ Goals: no other task in Goals precedes t}
  WHILE ( Goals is not empty ) {
    t ← choose nondeterministically some task in tasksAvailable that is not in pendingTasks
      // Backtracking point
    < primTask , subtasks > ← findReduction(S, SelectedTactic, t, D)
  IF (primTask is NULL) RETURN FAILURE // The task is either unknown or it cannot be handled
  ELSE IF (primTask is a primitive task) { // Backtracking point
    select < op , θ> so that op is a ground instance of an operator in D, θ is a
    substitution that unifies {head(op),primTask} and makes S satisfies preconditions(op)
    IF (θ is not successful) RETURN FAILURE // No operator matches this task
    ELSE { // This part executes the operator and updates the world state
      S ← S - deleteList(op) U addList(op)
      outputPlan ← append(outputPlan,head(op))
      Goals ← applySubstitution (Goals – { primTask } U subtasks , θ) constraining
        each task in subtasks to precede those tasks that t preceded in Goals
      handle "protection requests" for logical atoms
    }
  }
  ELSE IF (primTask is a built-in task) {
    IF (primTask needs to be answered by the user) {
      send primTask to todoGUIList for execution // Hook for tasks requiring mixed-initiative
      pendingTasks = pendingTask U { primTask }
    }
    ELSE { // The task has been answered/processed
      θ ← getResult(primTask, todoGUIlst)
      IF (θ is not successful) RETURN FAILURE // The user's answer was negative
      ELSE Goals ← applySubstitution (Goals – { primTask } , θ)
    }
  }
  // Those built-in tasks answered (or processed) are removed from pending status
  pendingTasks ← updateReadyTasks(pendingTasks, todoGUIList)
  // This part selects the tasks ready for the next iteration
  tasksAvailable ← {t ∈ Goals: no other task in Goals precedes t}
  } // end while
  RETURN (< outputPlan, S > )
} // End dbotsSHOP2

// It applies successive decompositions until finding the first primitive or built-in task
// whose execution will modify the world state
FUNCTION findReduction(S, Tactic, initialTask, D): < Task , Decomposition > {
  current ← initialTask  decomp ← Ø
  WHILE (current is not primitive or built-in task) {
    select < met , θ > so that met is a ground instance of a method in D, θ is a substitution
    that unifies {head(met), primTask}, met aligns with Tactic and makes S satisfies
    preconditions(met) // Backtracking point
    IF (θ is not successful) RETURN FAILURE // No method matches this task
    ELSE { // This part executes the method and computes a decomposition for the task
      decomp ← decomp – { current } U body(met) constraining each task in body (met) to
        precede those tasks that current preceded in decomp
    }
  }
  RETURN (< current, decomp >)
} // End findReduction
```

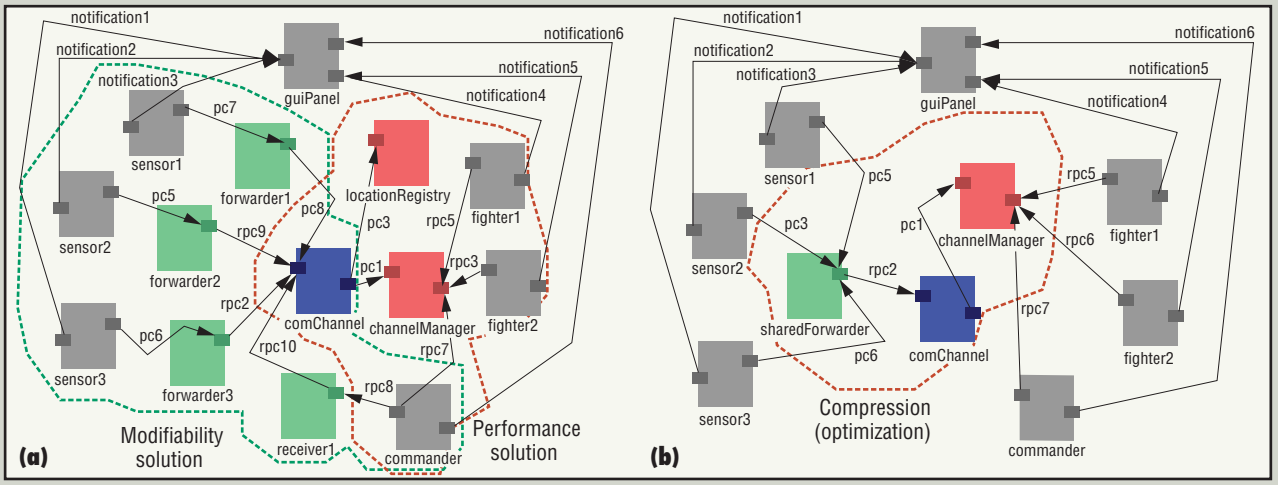
Figure 5. The HTN planning algorithm used by the designbots. The main part of the algorithm (at the top) handles the goals (tasks) that are directly achievable by HTN operators or the user-query tasks. The second part (at the bottom) decomposes nonprimitive goals by means of HTN methods.

that an architect (or a computer) can achieve the goals sequentially, in any order. This assumption relies on the notion of *quality drivers*.[1] When selecting a transformation, the mediator will prefer plans associated with scenarios marked as drivers over the rest of the plans. Figure 6a shows the results of this strategy in BCS, applying the modifiability and performance solutions sequentially. Although transformations aren't always commutative, this strategy performs well and is straightforward to implement. However, this strategy has a drawback: early commitment to a solution that focuses on a specific quality can hinder the consideration of better solutions later (even with backtracking).

The second technique is to merge groups of plans for different goals. This technique relies on a plan-merging heuristic[6] based on special categories of interactions among the actions in the designbots' plans. Unlike the previous strategy, the mediator constructs several equivalence classes to combine plans for different qualities into a joint transformation. Figure 6b shows the BCS architecture after the mediator merged the modifiability and performance solutions. According to the table in Figure 6, this strategy is closer to what architects do when faced with trade-offs among solutions. Nonetheless, the heuristic is more complex to implement (for example, identifying interactions is domain specific), and it doesn't guarantee a successful combination of plans. If this happens, the mediator might replace specific plans by asking a designbot for another plan for the same goals.

Evaluation

As a proof of concept, we developed a DesignBots prototype and conducted design-related case studies. The main objective was to demonstrate that we can model PAD with a planning approach. The successful criterion was then to exercise the search engine by generating a set of solutions with trade-offs. Because the focus was the framework itself rather than the amount of architectural knowledge embedded in the agents, we specified tactics and patterns to the extent necessary to produce different solutions. At this stage, rather than running experiments with architects using the tool, we performed a postmortem analysis in which the case studies provided material to reproduce design situations with DesignBots. That is, initial architectures and scenarios fed the tool, and we configured the



Scenario	Initial estimations	Sequential transformation	Merged transformation
1	Average cost change = 55%	Average cost change = 38%	Average cost change = 42%
2	Average throughput = 10% Average latency = 0.65 msec	Average throughput = 12% Average latency = 0.61 msec	Average throughput = 18% Average latency = 0.45 msec

Figure 6. Application of modifiability and performance solutions for BCS: (a) sequential and (b) merged. The two solutions have different trade-offs regarding the scenarios' performance and modifiability responses, as shown by the cost, latency, and throughput figures in the table.

agents with adequate tactics and patterns.[1,4] Then, we compared the DesignBots solutions against the human designs.

We built the prototype on top of a Java framework that permits the construction of designbots with planning and architectural design capabilities. An ADLite toolkit supports the visual editing of components, connectors, responsibilities, and scenarios. As for planning, we provided default implementations of both the dbotsSHOP2 and merging algorithms. On this basis, we carried out three case studies involving modifiability and performance concerns. Initially, the BCS case study allowed us to deploy and tune the DesignBots infrastructure. Then, we applied DesignBots in a moderate-size project for a software design course: we asked graduate students to produce solutions for the design of a home-alarm monitoring system (HAS). Finally, in the third case study, we tested DesignBots in the context of a telecommunications project as part of consulting activities made for the company Delsat Group.

As a sample of the case studies, the HAS case study comprised three modifiability and three performance scenarios, which were assigned to six different designbots. When exploring alternatives, a notable design aspect was the recommendation of a blackboard to coordinate action tasks, diagnosis, and high-level policies.[4] Among other transformations for the HAS architecture, the designbots helped us in architecting for support for adding new sensor types, configuration of reactive and diagnosis functionality, timing issues for reactive actions, and personalization of action rules. For the HAS case study, Table 1 (see p. 76) summarizes the architectural mechanisms specified for the planning domain and which of them were actually selected by the designbots to achieve the scenarios. The two rightmost columns of the table show a qualitative evaluation of the solutions applied by the designbots. The most relevant optimization during merging was about relaxing the constraints of the blackboard style in order to apply a blackboard variant that accounts for performance issues. (The original blackboard style is good for reducing rippling of changes but may have negative consequences on execution times and task priorities.)

Deciding the best way to write tactics and patterns was a central concern for the planning domain. In general, the HTN formalism admits many implementations of the same concepts. This depends on issues such as modularity, the level of the architect's intervention, or default values, among others. Because a clear task decomposition helps the architect visualize the relationships of tactics with patterns, we preferred to codify the domain as modularly as possible. Furthermore, opportunities for the architect's intervention were included only when this would avoid extra planning work. Figure 7 (see p. 77) presents planning data gathered from the case studies.

The planning engine was very useful in finding variants for a base solution the architects were familiar with and then improve that solution's quality-attribute measures. These variants showed structural similarities with those developed by people. Of the patterns applied by the designbots, the architects supervising the case studies considered 70 percent correct. We observed small differences in component configurations when compared to those arranged manually by architects. Although not a fundamental limitation of the planning approach, the fact that ADLite covers only a structural view of the architecture somehow restricted the analyses and transformations

Table 1. Alternatives generated for the home-alarm monitoring system case study.

Scenarios	Main design issue	Architectural tactics and patterns available to the designbots	HTN planning system: Supported	HTN planning system: Suggested	HTN planning system: Choice	Response analysis*: Sequential solution	Response analysis*: Merged solution
M1	Support adding new types of sensors within the device layer.	1. Separate the sensor interface from its implementation	Yes	Yes	Within option 2, the first mechanism was selected.	+	+
		2. Insert an intermediary between the devices and the data they produce or consume - **AbstractDataRepository** - DataIndirection - PublisherSubscriber	Yes	Yes			
M2	Configuration of reactive and diagnosis functionality should be easy for the user.	1. Provide customization of devices and their interactions - **PublisherSubscriber** - Façade - ClientDispatcherServer	Yes	Yes	Within option 1, the first mechanism was selected.	+	+/–
		2. Defer binding time - ConfigurationFiles - UniformProtocol	No	No			
M3	New configuration rules should be made available for the devices.	1. Provision of some kind of interpreter - **RuleBasedEngine**	Yes	Yes	Option 1 was the only available one.	+/–	–
P1	Fulfill the deadlines associated with the production and consumption of data.	1. Define scheduling policy - **PriorityBasedDispatcher** - RoundRobinScheduling	Yes	Yes	Within option 1, the first mechanism was selected.	+/–	+
P2	The level of response should be kept bounded.	1. Define scheduling policy - **PriorityBasedDispatcher** - RoundRobinScheduling	Yes	Yes	Within option 1, the first mechanism was selected.	+/–	+/–
		2. Manage event rate - NotificationDispatcher	Yes	No			
P3	The vocabulary of notifications can be updated, but maintaining the above level of response.	1. Define scheduling policy - **PriorityBasedDispatcher** - RoundRobinScheduling	Yes	Yes	Within option 1, the first mechanism was selected.	+/–	+

Trade-off

Trade-off

Trade-off

*Symbols +, –, and +/– reflect the relative variations in the scenario responses when applying two solutions, using the sequential and merged strategies, respectively.

derivable from it (for example, ADLite can't model a dynamic architectural view for performance).

Analyzing the HAS and Delsat case studies, we found that the planning engine elaborated more valid solutions for the HAS architecture and provided fewer and flawed solutions for the Delsat architecture. In addition to these case studies' different domain and problem sizes, the level of expertise required for each case study's design explains that finding. The former case study involved novice architects, whereas the latter involved experts in telecommunication systems. Even after equipping the designbots with sufficient tactics and patterns, they weren't always capable of emulating design experience. To deal with this technical hitch, the planning engine must incorporate more heuristic knowledge from experts.

The alternatives that the designbots built clearly influenced the quality-attribute analyses that set the goals. The relationships between architectural structures and qualities weren't always well reflected in the human designs. Because PAD is at the framework's core, an advantage of DesignBots is that it makes those relationships more visible in the resulting designs. The "constructive" procedures of HTN planning and the possibility of backtracking are two factors that sustain this assistance. Nonetheless, owing to the limited GUI and the lack of design

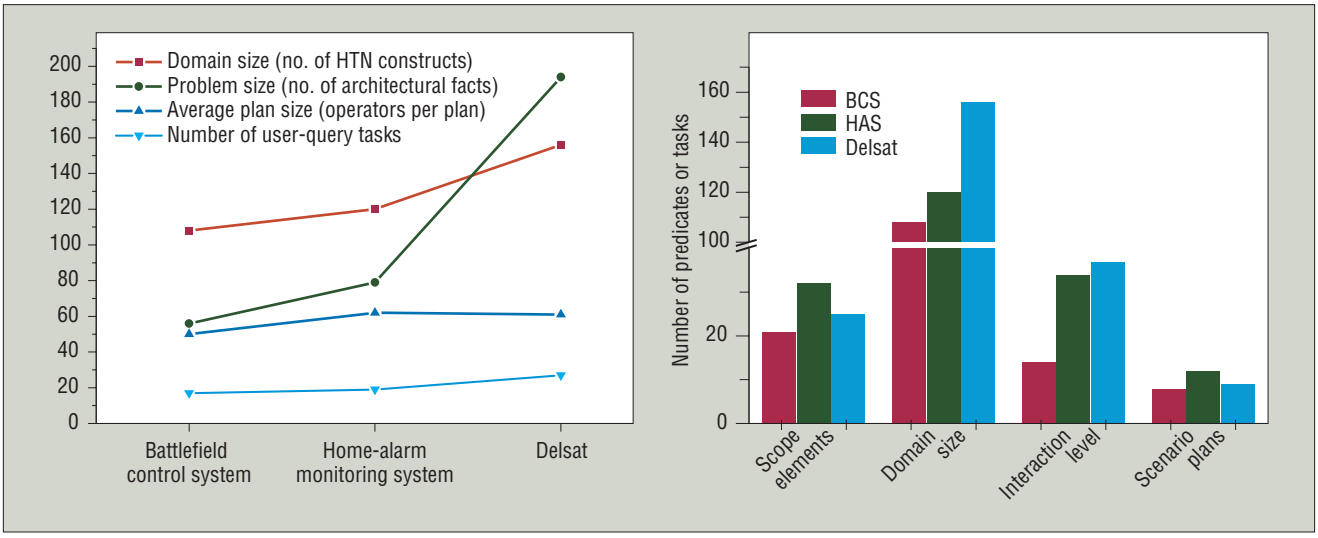


FIgure 7. Planning data gathered from the home-alarm system case study. Using the problem and domain specifications, the graphs give an idea of the complexity of the planning process. About two-thirds of the valid solutions were oriented to modifiability. The architects considered only three or four of these solutions acceptable.

rationale about what the planning engine was doing, the tool demanded considerable intellectual effort from the architects to understand the exploration.

Software architects are good at producing creative solutions, and they often use their experience to decide on quality-attribute trade-offs. On the other hand, recent advances in both planning and architectural techniques enable automating design tasks that had been reserved for humans. Planning-based assistants can augment architects' capabilities by reminding them about design constraints, pointing out promising directions to explore, and attending to implementation details of architectural patterns. Specifically, the contribution of the HTN planning model proposed for DesignBots is that of considering quality-attribute goals as drivers for the process in order to represent architectural tactics and patterns as hierarchical procedures whose function is to meet these goals.

Overall, this work reinforces the argument that AI-based tools can facilitate the design of architectures driven by quality-attribute issues. To enable the adoption of such tools, we must develop better control strategies and heuristics to guide the selection and instantiation of architectural patterns. Furthermore, we need empirical data about tool performance and users' acceptance. Provided that support is present, the integration of agents with planning techniques holds promise for a new generation of tools that can make architectural design less complex and more productive.

The Authors

J. Andrés Díaz-Pace is a member of the technical staff at the Software Engineering Institute, Carnegie Mellon University. His research interests include quality-driven architecture design, AI techniques, automated design tools, and architecture-based system evolution. Díaz-Pace received his PhD in computer science from Argentina's UNICEN University. Contact him at adiaz@sei.cmu.edu.

Marcelo R. Campo is a professor in the Computer Science Department and head of the ISISTAN Research Institute, both at UNICEN University. He is also a research fellow at Argentina's National Scientific and Technical Research Council. His research interests include intelligent tools for software engineering, software architectures and frameworks, agent technology, and software visualization. Campo received his PhD in computer science from Brazil's Universidade Federal do Rio Grande do Sul. Contact him at mcampo@exa.unicen.edu.ar.

References

1. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice,* 2nd ed., Addison-Wesley, 2003.
2. F. Bachmann et al., "Designing Software Architectures to Achieve Quality Attribute Requirements," *IEE Proc. Software*, vol. 152, no. 4, 2005, pp. 153–165.
3. D. Nau et al., "SHOP2: An HTN Planning System," *J. Artificial Intelligence Research*, vol. 20, 2003, pp. 379–404.
4. F. Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.
5. R. Kazman, M. Klein, and P. Clements, *ATAM: Method for Architecture Evaluation*, tech. report CMU/SEI-2000-TR-004, Software Eng. Inst., Carnegie Mellon Univ., 2002.
6. Q. Yang, *Intelligent Planning: A Decomposition and Abstraction-Based Approach*, Springer, 1997.