Appl Intell (2008) 29: 56–78
DOI 10.1007/s10489-007-0081-z

Experiences with planning techniques for assisting software design activities

J. Andrés Díaz-Pace · Marcelo R. Campo

Published online: 4 August 2007

Abstract Any software design process can be seen as a workflow of design tasks, in which the developer makes different decisions regarding both functionality and quality-attribute properties of the intended system. However, ensuring the right compromises among design solutions is always a problematic and error-prone activity for the developer. Furthermore, he/she may get overwhelmed by the diversity of design techniques and technologies nowadays available. Along this line, we believe that it is possible to provide a declarative representation of this knowledge, so as to apply AI techniques when searching for solutions in the design space. Specifically, mixed-initiative planning algorithms are an interesting approach to automate some aspects of that search. In this paper, we report on three successful experiences with planning to assist the developer in decision-making for architectural and object-oriented design contexts. We also analyze the perspectives of a planning-based approach in the development of software design tools.

1 Introduction

During the last two decades, the development of intelligent tools for supporting different software development activities has been an active field of research. Intelligent support aims at enhancing the productivity of developers, allowing the automation of repetitive tasks, but most importantly yet the reuse of expertise. Expertise is essential for every activity of software development, but in the case of design it plays a crucial role that can define the success or disastrous failure of a project [7]. This is so, because design usually cuts across phases such as: requirements analysis, software architecting and object-oriented implementation.

At its essence, designing software is a matter of *decision-making* regarding both functionality and quality-attribute properties of the intended system. Design activities operate at different levels of abstraction, and so do design decisions. For example, while designing a software architecture we may evaluate whether applying a passive or an active repository as intermediary among components; in the case of an object-oriented framework we may decide to create specific subclasses and customize a general interaction protocol; and when implementing a particular object-oriented class we may decide to make it a singleton. All these kinds of design share a common characteristic: the decisions strongly rely on the *developer's expertise* to manage the compromises among different design alternatives.

Over the last years, we have been involved in several research projects of tools based on AI techniques [13, 17, 32], which are able to help developers to perform complex design tasks. In particular, many of these tasks entailed a search process, usually driven by developers' preferences and goals in combination with domain-specific constraints. This suggested the use of a novel approach based on *planning* [27, 31] to support design assistance. From this perspective, we see the common workflow of design tasks as a planning problem stated in terms of a space of "software artifacts" and "design operations", in which the developer is interested in particular solutions of this space. Here, there may

J.A. Díaz-Pace (✉)
Software Engineering Institute, Pittsburgh, PA 15213-2612, USA
e-mail: adiaz@sei.cmu.edu

M.R. Campo
ISISTAN Research Institute, Faculty of Sciences, UNICEN
University Campus Universitario, B7001BOO Tandil,
Buenos Aires, Argentina
e-mail: mcampo@exa.unicen.edu.ar

M.R. Campo
CONICET, Buenos Aires, Argentina

be various sequences of tasks that when executed allow the developer to get the same solution. Since a complete search of the "design space" can be barely afforded using computational means, different sources of knowledge are necessary to prune and direct the planning search. Furthermore, the developer should provide feedback to the planner in order to make the assistance really effective.

In this article, we report on our experiences with planning for automating design activities such as: exploration of architectural designs, instantiation of object-oriented frameworks and weaving of aspect technologies. The characteristics of these domains led us to adaptations of the classical planning framework. First, the specification of knowledge sources varies depending on the type of design or its level of abstraction (e.g., architectural design, object-oriented design, or implementation). Second, the planning algorithms require the consideration of special features (e.g., regressive vs. progressive strategies, hierarchical structure of operators, mixed-initiative, incomplete information, integration with object-oriented programming, etc.). Many times, these algorithms have to sacrifice formal properties (e.g., completeness, soundness, termination, etc.), in exchange of a better applicability to real-world situations. The case-studies presented in the paper show that the "constructive approach" of planning can be an effective (and rather unexplored) technique for managing software design tasks. Essentially, we have demonstrated that planning-based tools provide a more powerful approach than other rule-based tools discussed in the literature, as planning can help the developer to focus on the important design issues, thereby, improving his/her productivity.

The rest of the article is organized around 6 sections. Section 2 gives background information about architectural and object-oriented design, and explains then how to treat three specific design problems from a planning perspective. Sections 3, 4 and 5 are devoted to the description of three planning-based tools we have developed for those problems. Each of these sections discusses the context of the tool approach, the role of planning within the tool, the details/adaptations made when implementing the planning engine, and some lessons learned. Section 6 gives related work. Finally, Sect. 7 presents the main conclusions of the work.

2 Problem context

The view of software architecture and that of the object-oriented paradigm have differences in their level of abstraction and roles within the design process. On one side, the architectural models are good at abstracting away from implementation details, and serve developers to reason about the main design decisions involving quality attributes within the system [7]. These quality attributes refer to performance, modifiability, security, availability, etc. Speaking in terms of global structure, an architectural design prescribes a collection of components, connectors and responsibilities to organize a system at early development stages. In these designs, it is common to take advantage of different *architectural styles* [9, 36] such as: layers, blackboard, pipes and filters, broker, client-server, etc. Basically, a style is a general design solution based on components and connectors that can be reused across different domains. Many styles have been codified in catalogs, so that the developer can pick a solution and tailor it to fit application-specific requirements.

On the other side, the object paradigm conceives a system as a collection of objects interacting via message sending [21]. An object-oriented design is certainly more concrete than a software architecture, as the object-related abstractions are concerned with design or implementation. In terms of quality, the way responsibilities are assigned to different groups of objects is what determines the flexibility/reusability of the resulting system. For this reason, the target quality attribute of the paradigm is principally modifiability, albeit ramifications to other quality attributes may exist. Nonetheless, the paradigm is equipped with abstractions that allow developers to implement useful object-oriented structures and collaborations, as in the case of *design patterns* and *frameworks* [20, 21]. Analogously to styles in architectural models, an object-oriented framework for a given domain gives a semi-complete skeleton of classes and behavioral patterns that can be extended by the developer to build specific applications in such a domain. Usually, the design of a good framework applies several design patterns.

More recently, several researchers have agreed on that objects have difficulties for accomplishing a clean "*separation of concerns*". Typical concerns here can range from high-level notions like security and quality of service to low-level notions such as caching and buffering. Among the techniques for separation of concerns that look to increase the expressiveness of the object paradigm, the most representative approach is perhaps the aspect-oriented paradigm [19, 25]. This "post-object" paradigm argues that systems are better designed/programmed by separately specifying the various system concerns along with some description of their relationships. Then, by relying on mechanisms provided by the underlying technology, these concerns can be composed or "woven" together into a coherent design/program. As a result, aspect-oriented technologies offer new modular abstractions called *aspects*, and also provide mechanisms for weaving these aspects with a base application.

In this context, a combination of distilled knowledge on architectural styles and object-oriented frameworks is very advantageous, when compared to developing specific design

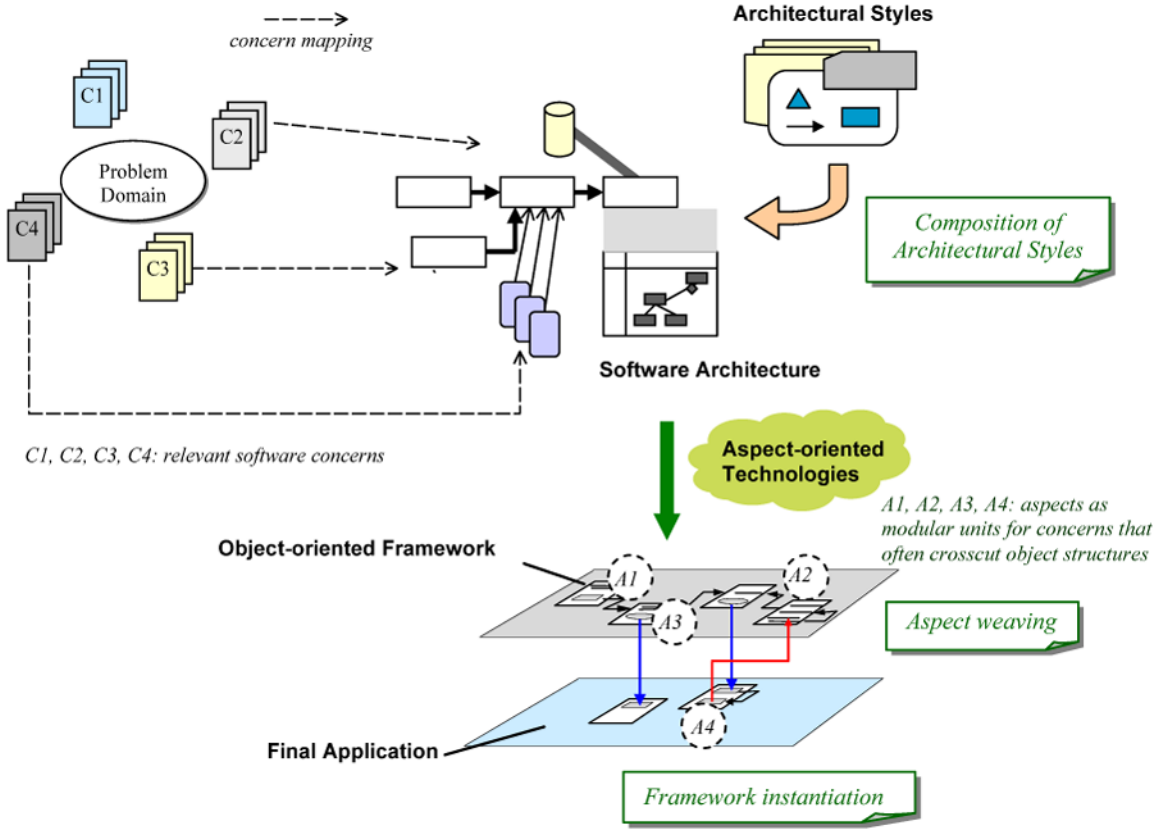

Fig. 1 Designing with architectural styles, aspects and frameworks

solutions on its own. First, understanding and applying predefined styles in the architecture of a system is likely to save time and assure quality. Second, implementing such a system on top of a framework for that architecture will enable the reuse of the object-oriented design, and eventually the code. Furthermore, aspects can help here to separate the expressions of multiple concerns in the framework, avoiding tangling of elements with each other and improving maintenance. This vision of design in which a software architecture is materialized through aspects and objects is illustrated in Fig. 1 [11, 12].

Unfortunately, taking advantage of all these approaches is not straightforward in practice, because certainly "the devil is in the details". According to our experience, some typical problems[1] developers are faced with when designing with software architectures, objects and aspects include:

(i) *Framework understanding and instantiation.* When using an object-oriented framework, the developer should first read the framework documentation, and then focus on how to materialize his/her application requirements in terms of the framework structure. This instantiation implies basically specializing some framework classes or providing implementations for certain methods. However, as frameworks implement very flexible design structures and make use of the so-called "control inversion" principle [20], they are complex and usually hard to understand. Therefore, building framework-based applications can be a difficult and time-consuming activity for many developers, especially for novice ones.

(ii) *Aspect weaving across different technologies.* When achieving separation of concerns in object-oriented designs, the developer should choose a specific aspect-oriented technology to support it. All the existent technologies somehow permit the encapsulation of crosscutting properties of the system as aspects, as well as the definition of join points to effectively link those aspects with application code. Nonetheless, there are different variants when coming to the weaving process, depending on the kind of implementation made by each technology [2]. This situation often requires an important grasp of experience and programming skills from the developer, who is responsible for providing the instructions to guide the weaving.

[1]The problems above have been presented following an increasing level of abstraction. That is, the problem of "framework understanding and instantiation" corresponds to detailed design (lowest level) while the problem of "composition of architectural styles" corresponds to conceptual design (highest level). The problem of "aspect weaving across different technologies" operates in between these two levels.

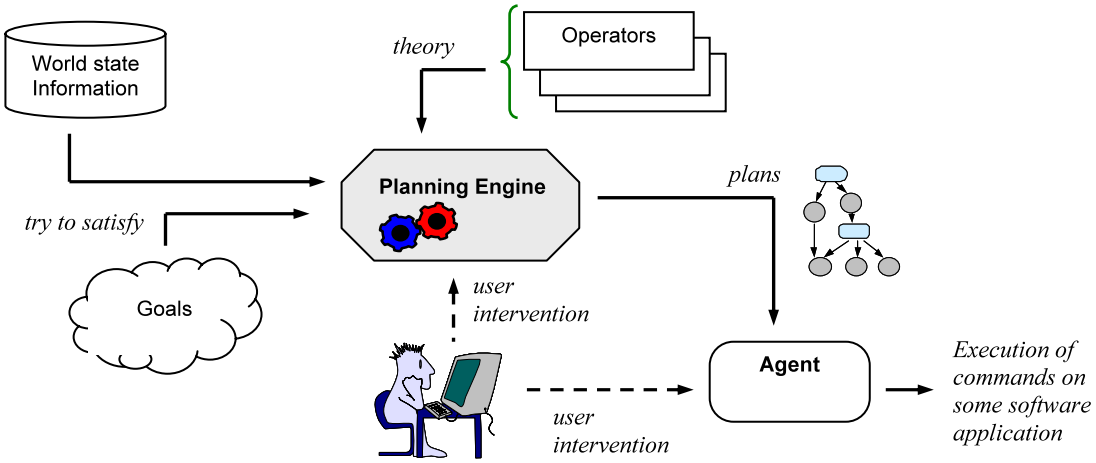

Fig. 2 Model for domain-independent planning tools

(iii) *Composition of architectural styles.* Given a set of quality-attribute requirements, we have multiple alternatives to design an architecture that satisfies those requirements. Consequently, an architecture is the result of a complex process of exploration, evaluation and composition of architectural styles, where specific styles naturally favor some qualities and hinder others. Styles are not applied randomly by the developer, but rather he/she is generally driven by guidelines encoding knowledge about good design practices. However, the number of available alternatives may get very large, even for applications with small design spaces. For this reason, exploring different styles and associated quality attributes in architectural design is usually a complicated, tedious and error-prone activity for developers.

2.1 The perspective of automated planning

The purpose of a planning system is to generate a plan, i.e. a sequence of actions that some agent can execute in order to achieve a set of goals [27, 42]. In the context of a planning-based tool, we assume this agent will make the translation of the plan into commands or changes to be processed by software applications. Basically, *planning is a trial-and-error search* in which many possibilities may have to be evaluated until finding a plan that works. To do so, there is a varied family of graph algorithms and heuristics such as: UCPOP, SHOP2, GraphPlan, or O-Plan [8, 30, 38, 41]. These algorithms differ in their features (e.g., information gathering, management of time and resources, or user interaction) and the types of planning domains they are designed to address (e.g., temporal, probabilistic or model-checking domains). Most of these algorithms are also domain-independent, in the sense that someone provides a particular problem to solve along with a domain theory, and the planning engine is general enough to deal with them. In this setting, we have usually an initial state and goals that constitute the *planning problem* along with a collection of operators that constitute the *planning theory*. Besides, the engine may provide opportunities of interaction to the user, in case additional information is needed while planning. A general schema of this kind of planning is shown in Fig. 2.

Interestingly, we can find a conceptual mapping between the "design strategies and tasks" carried out by developers and the techniques available to construct either architectural or object-oriented models. Instead of considering arbitrary paths of design, a developer usually starts with a guess of the solution (e.g., a conceptual application model or initial design) and sketches a strategy for a few, relevant objectives or "drivers" for his/her design [7, 37]. According to that strategy, the developer then evaluates and selects among the different solutions and variants valid for accomplishing the application requirements. Along this "strategic" search for solutions, the developer will make the key decisions for the design of the system. These decisions will move progressively towards more detailed decisions, until deriving a collection of actions that will directly modify (and refine) the components and responsibilities of the current design model. In a way, this workflow of decisions, tasks and actions can be though of as the *developer's design plan*, whose operators are symbolic representations of activities to be performed in order to arrive to a solution. Certainly, this view of design as a search is not perfect, and there are variations in practice depending on factors such as the developer's expertise, creativity, or familiarity with the technology. For instance, expert architects tend to apply familiar solutions even if these solutions are not optimal for the goals, as they understand their advantages and disadvantages. In the case of novice developers, they often prefer a narrow context of the problem and set of guidelines for its solution. In unfamiliar domains, the requirements can be explored by the developer at the same time as he/she tries to understand the space of possible design solutions [37]. Therefore, although the "design plan" provides a logical structure, the

choice of which goal is the following to work on can be seen as a *mix between opportunistic and prescribed design* [35]. Given these considerations, we analyze below how different planning techniques can approximate situations of architectural and object-oriented design.

Architectural design takes place at early development phases, and as such, it is characterized by abstractness and an incomplete understanding of the problem and/or solution. Nonetheless, there is a certain agreement among developers that the achievement of quality attributes comes mostly from using predetermined styles to organize the system [24] (e.g., client-server or fixed-priority-scheduler for performance, facade or blackboard for modifiability, firewall for security, etc.). These styles are actually ruled by design principles such as: abstraction, information hiding, modularization, separation of concerns, coupling, and divide-and-conquer, among others [9]. Thus, when an architect applies a transformation to an architectural model (in order to improve a given quality attribute), both principles and styles appear embodied in the transformation as a network of architectural operations. The transformations can explore different subjects at different levels of detail, following a reduction strategy via decompositions. The synthesis (or integration of the parts) is perceived as the main contribution by the architect himself. This type of transformational process fits well with the characteristics of a *hierarchical planner* [30].

In the case of object-oriented design, on the other hand, we follow a different design process than that for architectural models. Object-oriented applications are barely built from scratch, because developers usually rely on already-designed APIs, frameworks and technologies, in the hope of maximizing reuse and reducing implementation efforts. In this context of frameworks, the instantiation of an application requires the developer to perform a number of basic actions, namely: creation of specific subclasses, overriding of some methods, or configuration of groups of objects to follow a behavioral pattern established by the framework. All these actions are ultimately driven by the need of implementing specific functions required by the application. Normally, good frameworks provide different ways of implementing the same functionality. Therefore, considering these functions as goals, a *partial-order planner* could generate a list of "instantiation" actions for implementing an application on top of a given framework. This reasoning can be also applied to aspect technologies, because they are usually implemented in terms of object-oriented frameworks or languages[2] [2].

[2]We may have, however, some differences in the representation of the actions, because some actions necessary for aspect technologies often imply a higher level of abstraction than those for plain framework instantiation (e.g., composition of an aspect with functional code, weaving of all the aspects required by an application, etc.).

In addition, when planning for real-life applications, there are some decisions that cannot be made by the planning engine, simply because it is almost impossible to have complete knowledge or enough evidence to prefer one course of action to others. Here, a human can solve quite easily certain points of the planning process, although not always being able to have a general rule for his/her decisions. Because of this, some algorithms incorporate a modality called *mixed-initiative* [42] that allows a human user to provide specific inputs in the middle of planning. This means that plans are not generated at once, but rather they are constructed gradually through dialogs between the user and the planner. Another desirable aspect of planning algorithms is *partial ordering* [41]. Planning is said to be partial when the actions of the plans are ordered following the least-commitment principle. That is, we specify that an action must be executed before other action only when this is strictly necessary. The combination of mixed-initiative and partial ordering confers great flexibility to the planning process.

In the sequel, we describe three tools for design assistance that tackle the problems discussed in Sect. 2. The tools are based on a domain-independent model of planning, although each case-study presents its own variations and adaptations of the planner to the specific domain being solved. An important part of the evaluation of "experimental" tools is their usability and acceptance by users. Due to factors such as business pressures and other constraints, performing controlled experiments involving development teams is expensive and often impossible in the context of customer projects. For this reason, the issues discussed in this research are primarily qualitative rather than quantitative. We have performed an in-house evaluation, relying on feedback from experienced senior practitioners to asses the results of applying the different planning-based tools.

3 Case-study 1: active documentation for frameworks

As stated in the first problem, the benefits of object-oriented frameworks come at the cost of understanding the features and mechanisms offered by the framework to implement a particular application. This involves two issues: (i) how to effectively document a framework; and (ii) how to provide "wizards" to guide a developer when building framework-based applications.

To do so, we have developed the *Smartbooks* approach [32], which introduces the concept of "active documentation" as a mechanism to take advantage of the efforts of writing documentation for framework usage. In particular, framework documentation is specified in the form of *recipes* (or cookbooks) so that an automated tool can assist the developer in framework instantiation. At the core of the *Smart-*

books tool, there is a partial-order planner that works following a mixed-initiative modality.

From the point of view of a tool user, the *Smartbooks* approach proceeds as follows. Initially, the framework designer feeds the planning engine with a collection of instantiation rules adequate for the target framework. Then, the developer should focus on what the application is supposed to do. He/she basically identifies the main items of functionality intended for his/her application, with almost no knowledge of the framework internals. This functionality items will work as the goals of the instantiation process the engine must plan for. Once the tool generates an instantiation plan, the developer just executes (perhaps including some modifications) the list of tasks that compose the plan.

3.1 The *Smartbooks* tool

This tool consists of three main modules, namely: a UML editor, a task manager, and the planner itself, all of them originally implemented in Smalltalk.[3] We distinguish two types of actors: the framework designer that writes documentation for a target framework, and the application developer (i.e., the tool user) that takes this documentation for implementing the functionality of his/her application.

The UML editor serves to graphically specify the various parts of the target framework. According to *Smartbooks*, the framework designer should specify: the functionality provided by the framework, how this functionality is implemented by different components, and mechanisms to somehow constraint the way the framework should be specialized. This is documented by means of the so-called *instantiation rules*, which are a kind of recipes that capture the different ways of implementing predetermined pieces of functionality. These rules are based on a collection of *programming tasks* (e.g., class specialization, method overwriting, configuration of default policies, initialization, etc.) that normal developers carry out in order to correctly specialize a framework. Yet more importantly, these instantiation rules have a direct translation to planning operators.

As an example, Fig. 3 shows a graphical description of an instantiation rule for the *AspectModerator* framework [16] as well as the planning operator generated for that rule. The functionality covered by the rule is about the definition of a proxy for a component, where the main class should implement the *FunctionalProxyIF* interface and then wrap the component. Very briefly, the diagram prescribes the execution of four tasks to get that functionality, as follows: (1) the task *implementInterface* to produce a subclass of *FunctionalProxyIF*, (2) the task *defineVariable* that must add a *myComponent* attribute in the proxy class, (3) the task *defineMethod* that is in charge of overriding *Method()* in the component (to incorporate additional behavior), and finally (4) the task *optionalUpdateMethod* if any change in the constructor is required.

The task manager deals with the interaction between the developer and the planning engine, which is accomplished by means of tasks associated with the operators. These tasks may refer either to user queries or general framework issues. Examples of user-query tasks include: selection from a list of options, edition of text inputs, or execution of external tools. Examples of user-framework tasks include: creation of classes by sub-classing, implementation of abstract methods in sub-classes, or re-definition of base methods. In fact, the tasks listed in the body of the operator *createFunctionalProxy(ComponentProxy,Component,Method)* in Fig. 3 are instances of user-framework tasks. When processing an operator, tasks belonging to the first type are left pending by the engine, while the tasks belonging to the second type are stored as part of the final plan. The task manager has a GUI panel that presents all these tasks to the developer. During the planning process, the developer should answer those user-query tasks considered relevant for the application, and when the planning is finished, he/she must execute the user-framework tasks of the instantiation plan generated as output.

A task execution generally involves a GUI form (associated with the task type) with different parameters that the user has to fill in. For example, the execution of "new-class" task opens a form where the user specifies parameters such as class name, super-class name and documentation notes. The execution of user-framework tasks is translated to commands for the UML editor. These commands will update the class diagram of the application under construction with information about class inheritance, relationships, methods, etc. Although the tool is not meant to work as a code generator, the commands may generate code skeletons in which the user can fill in method behavior and implementation details. Furthermore, tasks can be classified either as optional or mandatory. This discriminates those tasks that are essential for obtaining a specific framework functionality from those tasks that, according to the documentation, are not indispensable for such a functionality. Optional tasks give information about programming activities that complement the tasks that must be executed.

3.2 The UCPOP planning algorithm

In framework instantiation, the rationale for applying least-commitment planning is the following. We know the initial state of the world (i.e., the framework structure) and we have

[3] The Smalltalk implementation of *Smartbooks* is not supported anymore. The prototype and the planner have been recently migrated to Java due to performance considerations. Therefore, the framework examples discussed in the paper usually reference Java-specific constructs such as: interfaces, abstract and protected methods, constructors, etc.

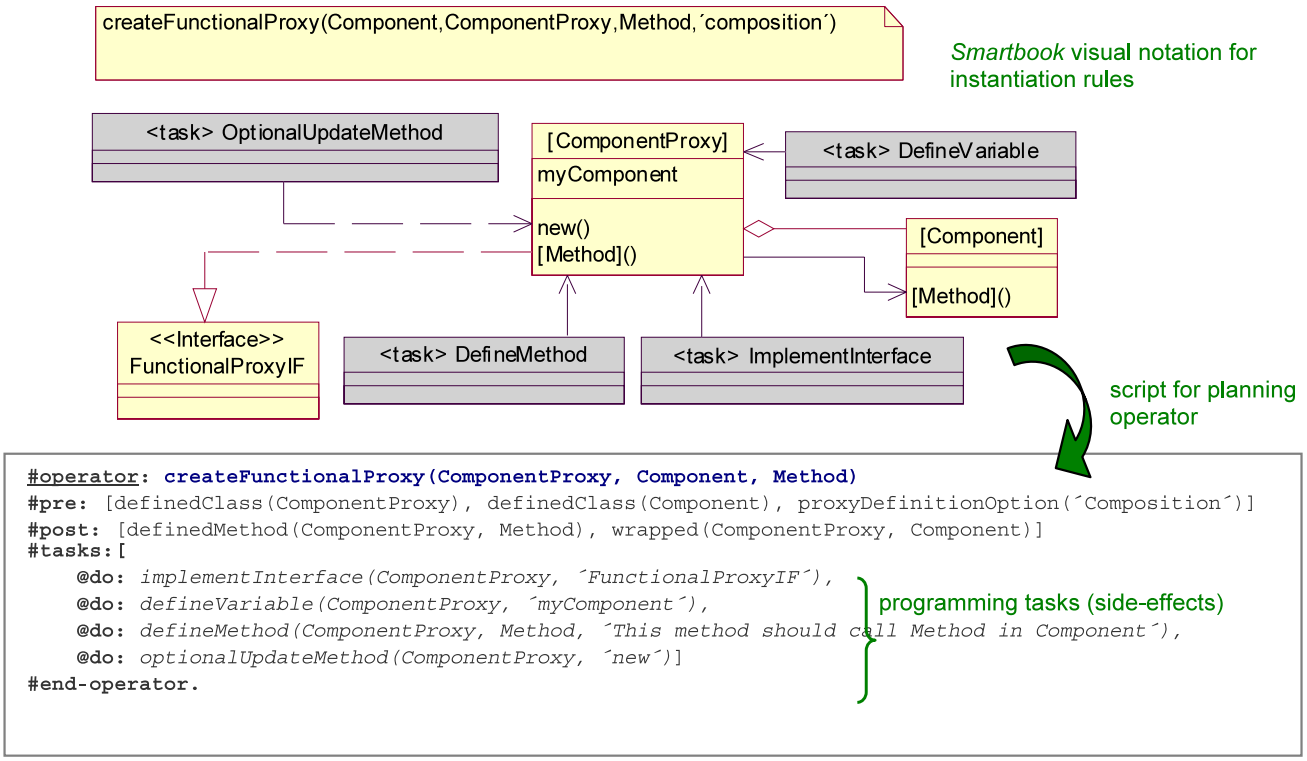

Fig. 3 A framework instantiation rule in *Smartbooks*

the goals clearly stated (i.e., which items of functionality should be delivered by the resulting application). However, we do not know the sequence of actions to reach the final state (i.e., the programming tasks to instantiate the application on top of the framework), which is precisely the job of the planning engine. As the developer is mostly driven by the goals (which are likely to coincide with the final state), it is reasonable to think about a regressive planning strategy. In addition, during framework instantiation it is customary to avoid making unnecessary decisions about tasks. For example, the tool should permit the execution of two tasks regarding class creation in any arbitrary order, unless some kind of dependency exists among them. This is a typical case of *least-commitment planning*. A planner that fulfills these requirements is the UCPOP algorithm [41].

Smartbooks uses a modified version of UCPOP, called *Hint*, that takes a set of functions for some target framework and elaborates a list of tasks required to implement these functions, based on a corpus of instantiation rules [34]. Although the basic *Hint* algorithm is codified in an object-oriented language, many planning features are supported by a Prolog engine embedded into the environment [1]. Internally, each instantiation rule is represented in the form of an operator with preconditions and effects, as shown previously in Fig. 3. An operator states which preconditions are needed for the effects to be true. The body of each operator is only evaluated when all its preconditions are true. If so, the tasks associated with the body are sent to the task manager. The algorithm is, basically, a loop that tries combinations of goals. At every step of the algorithm, the planner seeks to make true the preconditions of operators whose effects are linked to the goals. Therefore, the planner proceeds backwards, starting with the goals and tracing them back to the initial state. Preferences may be inserted for selecting certain operators that will likely make the plan succeed. In Fig. 4 we can see a partial representation of the UCPOP planning process used by *Smartbooks*. The plan includes a concrete action based on the rule given in Fig. 3. Further details of *Hint* can be found in [33].

The *Hint* algorithm applies backtracking, checking every combination of goals until producing a possible plan or returning an empty plan. Unlike traditional UCPOP, if a plan cannot be built for the original set of goals, the planner takes subsets and permutations of these goals. This happens when the framework documentation is not enough to completely describe how to implement all the functional requirements specified by the developer. This means that, despite having limited documentation, the planner can manage to generate some tasks for some of the goals, so that the developer can consider this aid in further refinements of his/her goals. One additional difference with UCPOP is that *Hint* allows the final plan to contain unbound variables, i.e. variables without an associated value. This feature is useful to leave some decisions to the framework user, e.g., the names of classes gen-

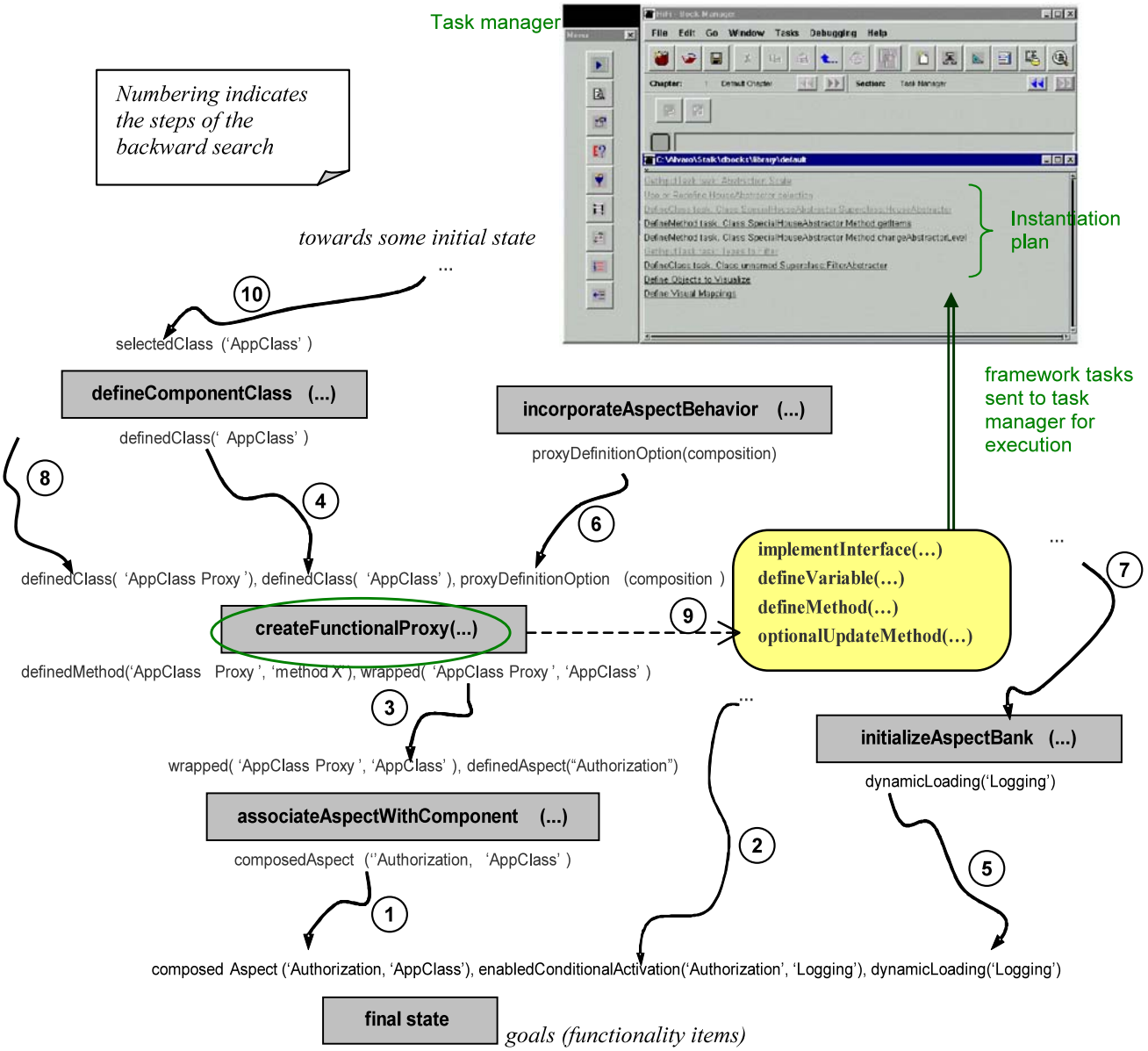

Fig. 4 Establishing links between operators within the *Hint* algorithm

erated during the instantiation process. The matching mechanisms of Prolog ensure that all the tasks in which a given variable is involved will reference consistent values.

However, instantiation rules are sometimes not enough for a correct specialization of the framework. For instance, the *AspectModerator* framework prescribes that a proxy must exist to add aspects to a functional class. This proxy must be implemented either by composition (as shown by rule in Fig. 3) or by inheritance. Despite these instantiation rules, the developer has freedom to specify functional classes and aspects on his/her own, but when the weaving time arrives, the framework will not activate the aspects unless the corresponding proxies are somehow provided. To account for these situations, *Smartbooks* is equipped with a set of consistency rules that complements the work of the planner by checking specific framework properties on the world state. A consistency rule is composed of three parts: the description of an inconsistent state that should be repaired; the configuration of tasks that can produce that state; and the tasks that can eliminate the inconsistency. A rule for the proxy-related inconsistency would say that if a functional class is crosscut by at least one aspect without a proxy, then a default application of the instantiation rule in Fig. 3 must be included in the plan. Some framework inconsistencies are domain-independent, like the attempts to create instances of abstract classes. As the plan is being developed,

the tool evaluates the base of consistency rules to detect possible violations to the framework prescriptions. If problems are encountered in the world state, repairing actions can be triggered and executed under the user's supervision. This consistency checking is again implemented in Prolog.

3.3 Lessons learned

The main contribution of *Smartbooks* is that it achieves an effective improvement on the guidance for framework-based application development. By centering the assistance on the application functionality, *Smartbooks* is especially useful with novice users, who generally have little knowledge about framework design. Nevertheless, the approach is flexible enough to assist more expert users too. Furthermore, if a user decides to ignore parts of the suggested plan, the instantiation rules can still guide him/her to implement the application consistently with the framework design.

At this moment, the *Smartbooks* tool has been tested with small and medium-size frameworks, such as: *AspectModerator*, *HotDraw*, *Luthier* [10, 23]. *HotDraw* is a simple graphical framework (∼30 classes) for implementing drawing editors in Smalltalk. The design of *HotDraw* was explicitly documented in terms of design patterns by its developers [21]. This characteristic much facilitated the codification of instantiation rules, because most of the framework designer's work was already available. In this context, we used *Smartbooks* to support the instantiation of a Pert editor, which is a default example included in the *HotDraw* distribution. *Luthier* is a framework created by one of the authors to support the understanding, exploration and visualization of object-oriented programs (and even frameworks) in Smalltalk. The general framework is quite complex (∼60 classes) and consists of four sub-frameworks, Two sub-framework were documented with the *Smartbooks* method, namely: *LuthierMOPs* and *LuthierBooks*. *LuthierMOPs* implements computational reflection mechanisms to gather information about objects at runtime, while *LuthierBooks* provides a model to represent that information in structured booklets based on hypertext. In this experiment, *Luthier* was used to help four groups of programmers to understand *HotDraw*, and then we validated that the results were in agreement with the *Luthier* design intent. Two of these groups instantiated *Luthier* on top of the *Smartbooks* tool, and they evidenced a quick progress in the development when compared to the other groups. On the other hand, we applied *Smartbooks* to a few sub-frameworks of a *CORBA* implementation, trying to ease the deployment of a distributed object-oriented application. Despite the framework's size and complexity, the results of all these case-studies have shown that the approach can be scaled up, because the instantiation tasks managed by *Smartbooks* are ultimately subclassing and method implementation [33, 34]. This is further supported by the capability of expressing implementation alternatives as separate instantiation rules, which enables the modularity of the documentation.

Among the drawbacks of the approach, we can mention the additional burden for the framework designer. Any documentation technique implies efforts from developers, but in this case, the more complete the instantiation guidance to provide the more details about the framework structure must be specified. The fact of writing the framework documentation using design patterns sometimes reduces the gap of codifying corresponding instantiation rules. Our experiences above corroborated that framework-based development is a hard and time-consuming task, and its success is highly tied to framework usability when constructing applications. For instance, a complex framework such as *Luthier* proved difficult to instantiate correctly without a deep study of its design documentation. Thus, it is worth investing the effort of producing good framework documentation, and tools like *Smartbooks* are an interesting spin off in this process. Regarding the planner, we observed that the time consumed by the algorithm grows exponentially with the size of the input domain. Although this time will never be significant when compared to the time involved in a development process, we still continue analyzing heuristics to optimize the behavior of the algorithm as the number of rules and goals increases. A possible heuristic here is to store plan fragments and reuse them in the satisfaction of similar goals. A related concern here is how to guarantee that the algorithm halts. Currently, the prototype has been ported to Java, in order to make some improvements to the planner and integrate the approach with CASE tools.

4 Case-study 2: exploration of design alternatives

When it comes to the third problem, the design of a software architecture that is "good enough" for a set of requirements is a difficult and challenging problem, even for experienced architects [39]. This is so because there are multiple alternative solutions that satisfy the same requirements, and each of these solutions may have different tradeoffs regarding quality attributes. Here, the notion of tradeoff means that the improvement of one quality attribute comes at the cost of the degradation of another [7] (e.g., modifiability versus performance, performance versus security, etc.). Because of that, architectural design can be seen as a complex process of exploration and analysis of design solutions.

In this context, we have built a tool called *Designbots* as a *design assistant* [17, 28] that can help the architect explore different alternatives for a given architectural design, evaluating whether the decisions that have been made are the right ones for his/her quality-attribute requirements. As the architect normally is in control of the leading design decisions,

this assistant is useful to handle the subsidiary decisions and routine tasks derived from the architect's principled decisions. Furthermore, the assistant is expected to keep track of alternatives, analyses and rationale.

In our approach, qualities do not arise spontaneously from the architecture, but rather they are planned for by the architect [7]. We take advantage of the relationships existing among quality-attribute issues and architectural styles for addressing these qualities, conceiving the construction of designs as a quality-driven searching process [17]. In particular, architectural design will involve a number of architectural transformations to achieve specific quality-attribute goals, where the transformations would capture architectural styles and the goals would represent quality-attribute scenarios [15, 24]. Assuming that a guess of the initial architecture exists, a planning-based tool can then suggest alternatives of improvement for that architecture. This way, the exploration of the design space is assimilated to a planning problem.

4.1 The *Designbots* tool

Designbots is a multi-agent object-oriented framework implemented in Java. This framework is based on the division of the design knowledge into a set of agents [28], which are capable of processing architectural information and proposing design transformations for a system in order to improve specific quality attributes. This kind of agents is called *designbot*. The multi-agent modeling aims at capturing the separate bodies of knowledge existing in the software architecture community regarding quality-attribute analysis and design. In more detail, each *designbot* is equipped with a *collection of architectural tactics, styles and mechanisms*, which constitutes the "architectural theory"[4] or planning domain of the agent. Also, an initial architecture and a list of quality-attribute scenarios for this architecture are entered by the architect. After being assigned to a group of scenarios, each *designbot* is able derive a number of quality-attribute goals from them. Both the initial architecture and these goals define the *planning problem* the *designbots* have to plan for. For doing their work, the *designbots* rely on a *hierarchical planning engine*. Figure 5 depicts the relationships between our architectural theory and the planning framework.

A transformation suggested by a *designbot* is actually a design plan, resulting from a single-quality analysis of the architecture (e.g., a performance-based plan, a modifiability-based plan, an availability-based plan, etc.). A transformation generally combines high-level architectural styles (e.g., like those discussed in catalogs [9, 36]) and lower-level architectural mechanisms (e.g., a cache for performance, a data router for modifiability, a firewall for security, etc. [24]). In order to account for tradeoffs among solutions, the framework also incorporates an agent called *mediator* that is responsible for coordinating the *designbots*' activities. This means that the *designbots*' goals are prioritized, balancing the effects of individual plans on the redesign of the architecture. Once a set of candidate transformations is ready, the architect can select any of these transformations and proceed to modify the base architecture.

Currently, the *Designbots* tool provides support for the following activities: edition of architectural models and quality-attribute scenarios; analysis of architectural models based on quality-attribute parameters; specification of tactics, styles and architectural mechanisms; and execution of the planning engine as well as merging of resulting transformations. To do so, the prototype comprises three main modules besides the planning engine, namely: a toolkit for architectural descriptions, a multi-agent infrastructure and a library of analysis models for quality attributes. Very briefly, the toolkit is used to create and manipulate software architectures. We have a custom ADL (Architectural Definition Language) with a vocabulary of systems, responsibilities, components and connectors. Components describe elements that have some kind of runtime presence (e.g., processes, clients, servers, data stores), while connectors capture the pathways of interaction between components (e.g., message passing, events, access to shared storage, etc.). In addition, all these elements can be annotated with properties. Regarding quality-attribute scenarios, they are initially expressed in a textual form. During analysis, each scenario is processed by the corresponding *designbot* (with help of the architect) to extract the main responsibilities implied by the text. For determining a set of goals from a given scenario, the tool is equipped with analysis techniques that allow the architect to estimate quality-attribute values for the current architecture. To obtain a representation of the world state on which the planner will perform its computations, the tool exports ADL specifications to a collection of Prolog facts. The quality-attribute goals of a *designbot* are translated to a hierarchical task network.[5] Finally, the multi-agent infrastructure provides most of the machinery for the *designbot* and *mediator* agents. A more detailed description of the approach is given in [17].

In order to support interaction with the architect, the planning engine implements the metaphor of user-query tasks already described for *Smartbooks*. This way, the architect is aware of the main design decisions, and he/she can opportunistically answers critical points of the solution being

[4] Much of the architectural design knowledge for *Designbots* is taken from the PAD procedure [6] developed at the Software Engineering Institute.

[5] Both the world state and the goals supplied to the planner are basically first-order logical predicates.

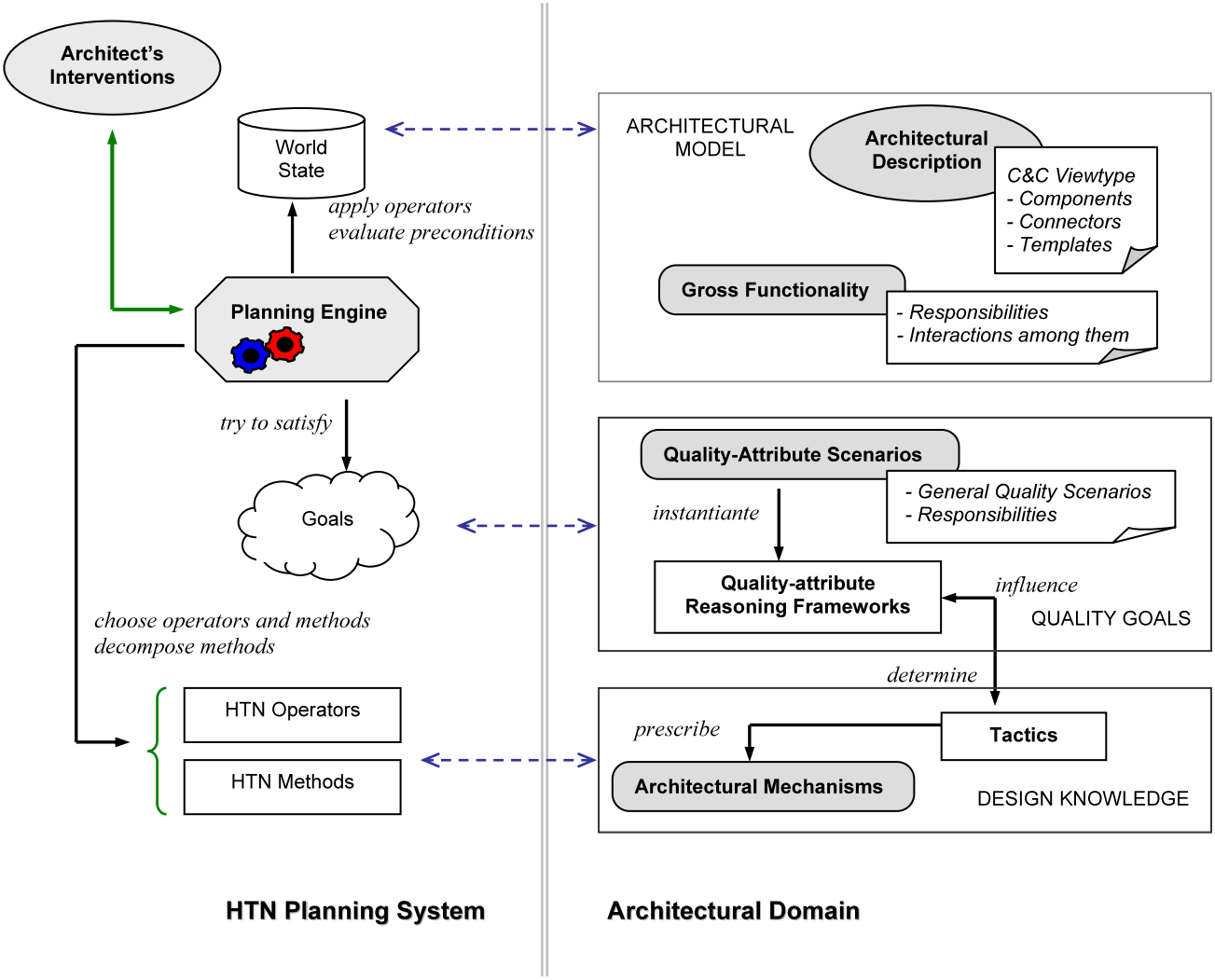

Fig. 5 The *Designbots* approach and its mapping to hierarchical planning

built [42]. Examples of user tasks for the architectural domain include: selection or creation of components, allocation of responsibilities to components, links among specific components and connectors, etc. Internally, however, these tasks are treated slightly differently than in *Smartbooks*, because now the planner is concerned with the decomposition of an input task network into lower-level tasks until finding a plan that can be directly executed on the world state.

4.2 The SHOP2 planning algorithm

Casting architectural design to planning brings around different requirements than those stated for the instantiation of object-oriented frameworks. In architectural design, we know the initial state (i.e., the components and connectors that shape the architectural model) but we have only an approximation of the goals (i.e., the desired levels of quality attributes for the improved version of the architecture). Here, the goals are "intentional" in the sense that they express a quality-attribute criterion that any solution (i.e., a configuration of components and connectors) should satisfy, instead of giving us the solution itself. The presence of tradeoffs among qualities may produce complex interactions among goals and related criteria. Normally, we do have neither the final state of the architecture nor the sequence of operators to reach that state (i.e., the resulting architectural transformation). Given these facts, the job of the planner is to perform a trial-and-error search and come up with candidate transformations.

In principle, the planner can produce transformations leading to re-designs of the original architecture that might or might not improve a quality of interest. A transformation will be valid for the architect as long as its evaluation does not stray outside of the range of accepted values defined by the quality-attribute analysis of scenarios. Because of this, the use of a backward search (like in UCPOP) turns out to

be problematic. Basically, a regressive strategy does not provide a suitable mental model for the architect to reason about where the planner is going to. Things get aggravated if the final configuration of the world state is unknown. Therefore, despite some performance losses, we consider that a forward strategy is a more rational choice for the planner. This direction of search allows us to represent the planning operators in a hierarchical way, so that the planner can analyze the feasibility of strategic solutions at an abstract level before providing the full details of their implementations. This style of planning is taken by the HTN (Hierarchical Task Network) planning model. A quite efficient algorithm for HTN planning is SHOP2 [30]. Essentially, SHOP2 uses a search-control strategy that performs an ordered-decomposition of task networks, generating the actions of a plan in the same order that they will be later executed. As a result, the current state is known at each step of the planning process, which makes it easy to reason about what is true and what might be true when planning.

In the HTN model, we have tasks that represent goals plus methods and operators useful for accomplishing the tasks. The planner starts with an initial network, and proceeds by applying methods that decompose each of its "abstract" tasks into simpler tasks until a plan consisting of primitive tasks is found. Unlike abstract tasks, for each primitive task, the planner chooses an applicable operator and instantiates it to produce an action that can be directly executed on the world state. If all of the constraints are satisfied, then the planner has found a solution plan; otherwise (e.g., dead-ends are encountered) the planner will backtrack and try other methods or other instantiations.

For the purpose of *Designbots*, the HTN methods and operators can be seen as "standard procedures" to capture the architectural styles and mechanisms used normally by architects when doing architectural design, while the tasks can be seen as the goals they have in mind when considering quality-attribute scenarios. Within the architectural domain, the HTN methods and operators can range from basic actions (e.g. creation/deletion of components or allocation of responsibilities) to more complex actions (e.g. delegation of responsibilities or insertion of a blackboard as intermediary). Moreover, some "higher level logic" is added to organize the application of these actions. This logic intends to capture abstract design strategies known as *architectural tactics* [7]. Unlike styles that often affect several quality attributes at the same time, a tactic provides directives for improving a single quality attribute. For instance, the tactic of "breaking the dependency chain" is relevant for modifiability, and the tactic of "balancing resource allocation" is applicable for performance. The tactical directives can be later fleshed through different combinations of architectural styles and mechanisms. In our framework, when a *designbot* selects one or several tactics (as a result of a previous analysis of scenarios), this enables corresponding HTN methods and operators that provide knowledge so that the planner can satisfy the *designbot*'s goals.

Conceptually, as the planner moves across a task network and decomposes its tasks, the most abstract tasks will typically capture design decisions (embodied by tactics) without concrete effects on the architectural model, while the tasks at the intermediate and lowest levels of the decomposition are likely to involve actions that modify the architectural model. In order to clarify this process, Fig. 6 shows a task network for two quality-attribute scenarios about modifiability and performance respectively. The goals have been inferred in the context of a particular quality-attribute analysis. This example shows the steps of planning for the task *checkDependencyChain*(...), as accomplished by a *designbot*. Note that, at planning time, the target task is tackled with a streamline of HTN actions that correspond to a particular tactic and associated styles and mechanisms.

Let's suppose the task *checkDependencyChain*(...) is a modifiability goal that involves breaking a dependency between two components, so that inserting a new component as intermediary is a suitable tactic for the goal. According to the type of dependency, this tactic could be implemented by different architectural styles, namely: a virtual machine, a repository, a naming server, a publisher-subscriber, a forwarder-receiver, etc. Let's consider here that the planner selects a *forwarder-receiver* [9] to break the dependency. This decision will trigger subsequent methods and operators (they are marked by green tasks in Fig. 6). The example additionally shows how "extra-planning" routines can be executed within the planning process. When the planner applies the method *checkDependencyChain*(...), the execution of the task @*selectOption*(...) will display a GUI panel with options and the planner will wait for the architect's decision. The architect should judge whether the dependency is problematic to be broken or not. By means of these built-in tasks, we incorporate mixed-initiative during the task decomposition. A task manager, like in *Smartbooks*, determines which tasks are pending for execution and which tasks have been already answered by the user. The latter tasks are returned to the network, enabling the planner to proceed with their successors.

The planning engine implements the basic features of SHOP2 as given by [30], although based on a seamlessly integration between Java and Prolog called *JavaLog* [1]. A nice aspect of *JavaLog* is that it exploits the advantages of both the logic and the object-oriented paradigms. Therefore, we can handle parts of the planning process that involve rules or backtracking with Prolog (e.g., evaluation of method preconditions over the world state, or derivation of goals and tactics), whereas those computations that are known in advance or involve significant costs can be programmed directly in Java (e.g., the core of the planning algorithm, or management of user-query tasks).

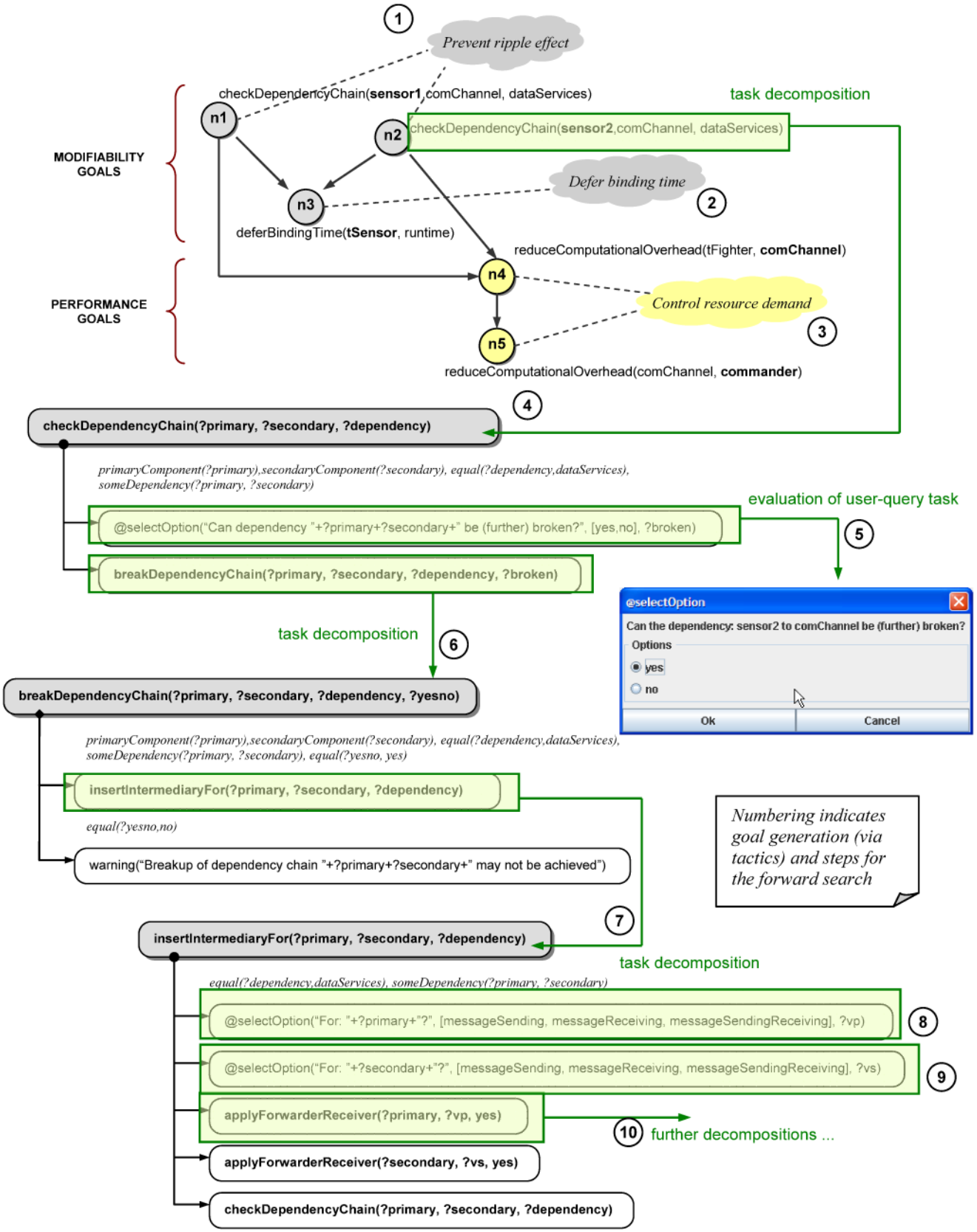

Fig. 6 Constructing a transformation for an HTN task network with quality-attribute goals

When the tool manipulates a design, the current architectural representation may conflict with the representations being analyzed by other *designbots*. As in *Smartbooks*, maintaining consistency within the world state becomes an issue. In fact, we have adapted the framework consistency rules to work with architectural specifications. An "observe-and-repair" procedure implemented in *JavaLog* takes care of any modification attempting to violate the architectural integrity, and let the architect know about this situation. Examples of inconsistencies that the *Designbots* tool tries to fix include: connectors that do not match the component type they are linked to, duplicate names, components with orphan interfaces, responsibilities with no parent components, etc.

4.3 Handling tradeoffs as multi-goal planning

So far, we have described how the *designbots* take quality-attribute scenarios and are able to produce transformations locally according to their own quality-attribute analyses. This is appealing in the sense that each *designbot* pursues one quality at a time; therefore, the complexity of the design is likely to be reduced. However, we know that interesting architectural decisions often have to do with multiple quality attributes. Therefore, it is necessary to handle both quality-attribute goals and transformations from a unified perspective. This aspect is addressed by the *mediator* agent at the planning level, by considering the *designbots*' solutions as a case of *multi-goal planning* [43].

Two techniques are used by the *mediator* agent to manage the *designbots*' plans. The first technique simply instructs each of the *designbots* to plan in some predefined order, relying on the notion of "architectural drivers" [7]. An architectural driver is a quality-attribute scenario that is influential for the design because shapes the architecture. The second technique, in turn, asks all the *designbots* to generate plans and then tries to combine them into a joint plan. This joint plan returned by the *mediator* is expected to reflect some tradeoffs among the characteristics of the individual plans.

On the planning side, the first technique states a condition of linearity among the goals of separate *designbots*, in the sense that their whole task networks can be solved sequentially by the planner in any arbitrary order [27]. Although linearity may be not always valid because the application of architectural patterns and mechanisms is not always commutative, this sequential technique performs well in many cases and is straightforward to implement using the multi-agent infrastructure. According to this, when selecting a transformation for the architecture, the *mediator* will prefer the plans generated by *designbots* whose scenarios have a high priority. The main drawback of applying solutions sequentially, even with backtracking, is that early commitments to a solution strongly focused on a specific quality can hinder the consideration of better solutions later on.

A more interesting approach is the one proposed by the second technique, supported by a plan merging heuristic described in [43]. Depending on which kinds of interactions occur among the tasks in individual plans, the heuristic constructs a number of equivalence classes and derive a joint plan. Three types of interactions are covered by the heuristic, namely: task-precedence, identical-task, and task-merging. The latter type of interaction is important for the merging process, because it allows us to define a composite task capable of accomplishing the "useful" effects of a set of tasks across different plans while leaving the resultant plan correct. Although some guidelines exist, determining the list of interactions for a given set of plans is usually a domain-dependent activity (e.g., one could detect a task-merging interaction by looking for tasks that contain sub-tasks that cancel each other out, as often happens among tasks that require common resources). The two remaining types of interactions are straightforward to compute. Currently, the construction of the list of interactions is managed by the *mediator*, under the supervision of the architect. Basically, the architect is initially asked to select plans as candidates for merging, then the mediator analyzes the plans and suggests interactions for their tasks, and the architect finally decides which of these interactions should be processed by the heuristic. Examples of rules for task-merging interactions that perform "optimizations" within the architectural domain include: replacement of 1-to-1 component relationships by a n-to-1 relationship, removal or insertion of facades, cancellation of unnecessary connectors, compression and separation mechanisms, etc.

Certainly, this second technique gets closer to what architects do when faced with tradeoffs among solutions, casting tradeoffs to a number of optimization over the plans for individual qualities. Nonetheless, the heuristic comes at the cost of more planning computation, and it does not guarantee a successful combination of plans when ordering conflicts arise. For example, when two tasks of different plans present a harmful interaction that cannot be solved (e.g., *a* before *b* in *plan1* and *b* before *a* in plan2), the merging of the plans is not possible. If so, sometimes the *mediator* can replace one of the problematic plans by asking a *designbot* to generate an alternative plan [43]. In a further more general treatment of tradeoffs, we expect the *designbots* will be able to reason about the implications of their plans and negotiate counter-proposals at planning time. To do so, the structure of the task network needs to be augmented with information about the decisions behind decompositions and revaluations. Here, we envision that a dependency-based backtracking schema can help to "repair" parts of the plans using predefined patterns, according to the actual tradeoff conditions and previous decisions.

4.4 Lessons learned

As a proof-of-concept, we have applied the *Designbots* tool in some small and medium-size case-studies [17]. In particular, three architectures with different modifiability and performance scenarios were studied, according to an "incremental" strategy. Initially, a case-study based on the simulation of a battlefield control system (BCS) served to deploy and tune the *Designbots* infrastructure. Having this ready, we applied *Designbots* to a moderate-size classroom project involving a home alarm monitoring system (HAS), where we had graduate students of a software design course produce alternative solutions for HAS. Finally, in the third case-study (DELSAT), *Designbots* was tested in the context of a telecommunications project, as part of some consulting activities made for the company Delsat Group. We considered two usually conflicting types of *designbots*: modifiability-centered and performance-centered agents. Each type of *designbot* was equipped with assets such as: a quality-attribute analysis models, tactics and related architectural patterns and mechanisms. After reproducing various design situations with the tool, we performed an empirical comparison of the *designbots*' alternatives against the human designs. As a sample, some results of the BCS case-study are summarized in Table 1. BCS comprised 5 modifiability scenarios and 2 performance scenarios, which were assigned to 7 different *designbot* agents. Among the transformations undergone by the initial architecture, we can mention: configuration of sensor technologies, message flow control at the shared channel, separation of presentation aspects, support for variant message formats, centralization of mission data, platform portability, and interaction between order dispatching and information updating. In Table 1, we show the architectural mechanisms specified for the planning domain, and which of these mechanisms were actually selected by the *designbots* to generate solutions for the scenarios (symbols $+$, $-$ and $+/-$ reflect the relative variations in the scenario responses when applying two solutions, using the sequential and merged strategies respectively).

From the perspective of design assistance, the architectural alternatives recommended by the *designbots* showed in general satisfactory similarities with those manually developed by people. Approximately, a 70% of the patterns applied by the *designbots* were considered correct by the architects that participated into the projects (minor differences were observed in the configurations of components when compared to those arranged manually by the architects). In particular, these architects also found the prototype useful regarding the exploration and recording of alternative plans of action or variants for a base solution they were familiar with. The "constructive schema" supported by the planning paradigm and the knowledge representation amenable for quality-driven search were the two factors that enabled this kind of assistance.

Since the planning domain is the most relevant semantic knowledge the planner has access to, the "architectural design" domain had to be carefully specified. Deciding what is the best way of codifying architectural knowledge in HTN was a central concern we faced at this stage. The notion of task decomposition implicit in HTN helped us to directly represent the relationships among tactics, styles and mechanisms. However, the writing style for HTN methods and operators may admit many alternative implementations of the same concepts. This depends on issues such as: modularity of methods and operators, level of interaction with the architect, and inclusion of default values, among others. In this context, the HTN formalism put us into a tension between the intuitive ability of SHOP2 to exploit abstraction and the efforts to program the domain constructions [27]. Assuming that a clear decomposition of tasks helps to visualize the steps of the exploration, we preferred to codify the planning domain as modular as possible. Furthermore, opportunities of interaction for the architect were included only when this would avoid extra work in the planner.

Other analyzed factors were accuracy of the plans and scalability. Unlike traditional planning where the world state and the actions are fairly static and enclosed, planning for architectural design requires one to take advantage of whatever knowledge is available. Among the aspects that contribute to this, we can mention: large architectural descriptions with constraints, architectural tactics that often need a substantial collection of domain operators, and resulting plans so complex that can be not very intelligible for architects. For this reason, the startup with *Designbots* demanded considerable efforts, with many backward loops until we could produce a useful collection of architectural transformations. In exchange for this work, the tool was finally able to manipulate basic architectures for the case-studies and plan a variety of alternatives for them.

On the other hand, we found an interesting aspect, which was not initially foreseen, regarding the solutions proposed by *Designbots* with respect to the experience of the architects. Comparing the results of the HAS and DELSAT case-studies, we observed that the *designbots* elaborated better solutions for some case-study, while in other case-studies they provided somehow inferior solutions than those developed by the experts involved in the project. An explanation for this tendency is that the planner, unlike expert humans, is not always capable of quickly discarding alternatives perceived as deficient, because it cannot emulate design experience. Even when equipping the *designbots* with sufficient architectural knowledge, the SHOP2 algorithm does not have yet guidelines to select "promising" decompositions of tasks. Thus, as the planner searches for solutions, it cannot be aware of the context in which particular transformations may or may not be useful. This situation may lead to combinatorial explosion problems or unnecessary complex

Table 1. Design alternatives generated by *Designbots* for the BCS case-study

Scenarios	Main design issue	Architectural tactics and mechanisms available to the *designbots*	HTN Planning System			Response Analysis	
			Supported	Suggested	Choice	Sequential Solution	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		tradeoff	
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		tradeoff	
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	+/-	-
							tradeoff
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	+/-	+/-
		1. Manage event rate - *NotificationDispatcher*	Yes	No			tradeoff
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	+/-	+

configurations. These flaws were mainly due to the codification of the planning domain and the lack of design knowledge embedded into the planning engine.

5 Case-study 3: weaving of aspect-based applications

Nowadays, the aspect-oriented paradigm has reached a reasonable level of maturity, and a comprehensive collection of practices and techniques to achieve the principle of *separation of concerns* [19, 25]. However, each aspect technology comes certainly with its own features, benefits and drawbacks. Regarding weaving, we see that the procedures to combine aspect-related and application-related information often require the understanding of many implementation-specific details. Therefore, this factor still compromises the application of aspect-oriented techniques by normal or novice developers [18].

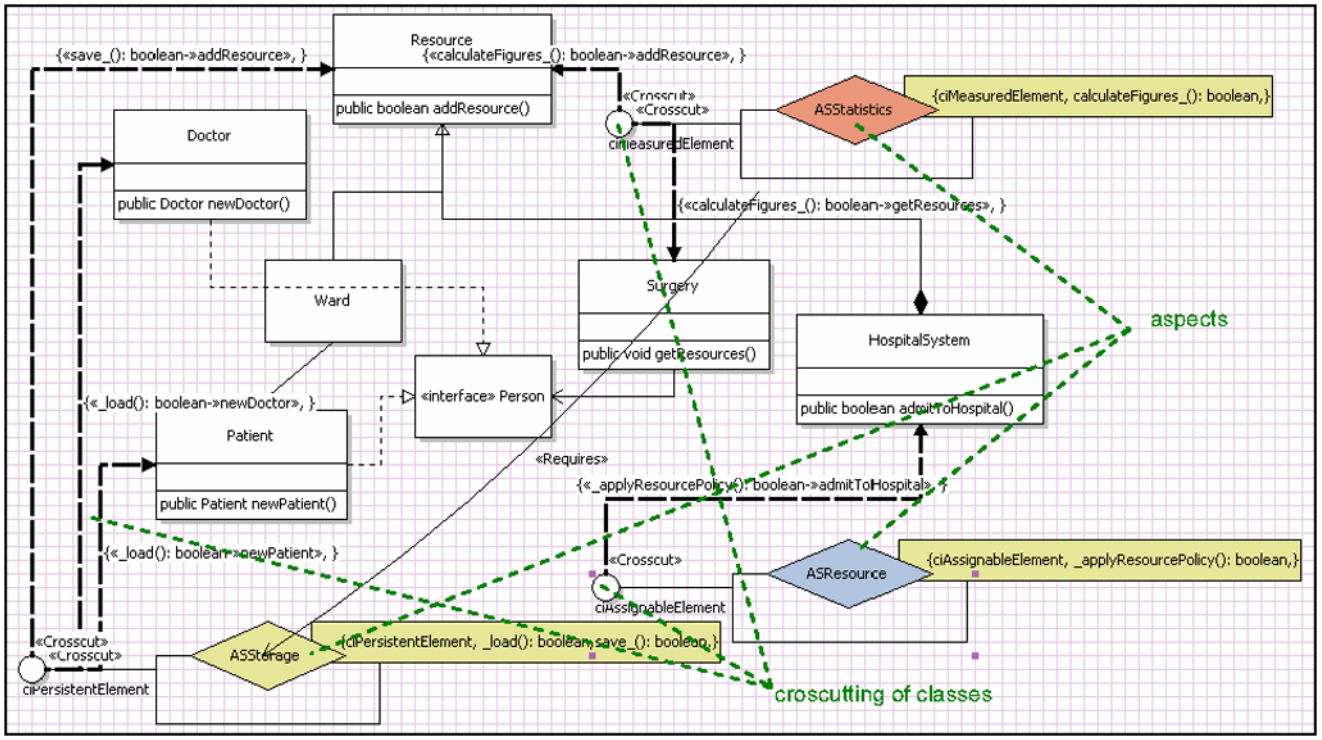

Fig. 7 AsideML notation to model aspect-oriented applications

In this context, the second problem about aspect weaving bears similarities with that of framework instantiation. We can see the aspect model as a general framework, involving the instantiation of a collection of "weaving operations" using different supporting technologies. These weaving operations will be ultimately mapped to concrete programming tasks. Based on the experiences with *Smartbooks*, we developed a tool called *Smartweaver* to help a developer to manage aspects in their projects, without being overwhelmed by the need of learning new and sophisticated aspect-oriented languages or frameworks all the time. The very idea of "*smartweaving*" uses planning to mediate between the developer's goals and the weaving mechanisms supported by the different aspect technologies (e.g., *AspectJ*, *AspectWerkz*, *AspectModerator* framework, etc. [4, 5, 16]). Furthermore, this case-study is interesting because it admits the two types of planning explained in Sects. 3 and 4. Initially, we implemented the *Smartweaver* planning engine with UCPOP, and then this implementation was gradually evolved towards SHOP2.

5.1 The *Smartweaver* tool

Basically, *Smartweaver* is a Java tool to support the weaving of aspects with functional components, assisting the developer in the implementation of that weaving across different aspect technologies. Along this line, the developer models his/her application in terms of classes, aspects and crosscutting relationships among them, by means of a formalism called *AsideML*. *AsideML* is a UML-like notation for aspect-oriented modeling that is neutral to different aspect technologies [40]. All this graphical information is then collected by a planning engine, which is provided with rules to translate the *AsideML* models to particular aspect technologies. The supporting technology can be either an aspect-oriented framework (such as the *AspectModerator* framework) or an aspect language (such as *AspectJ*).

The main benefit of this approach is that the developer is only concerned with general aspect-oriented constructs when developing his/her application, and once a specific aspect technology is selected, *Smartweaver* generates a list of tasks to carry out the weaving process with that technology. The execution of these tasks will turn his/her general design model into a technology-specific design model. The architecture of the tool comprises three main modules, namely: an *AsideML* editor, a task manager and the planner itself. During the planning process, like in *Smartbooks*, the developer can interact with the tool based on a mixed-initiative metaphor. This functionality is implemented by a corresponding task manager component. Figure 7 shows a structural aspect model for an application in *AsideML*. In addition to this, the developer can model behavioral views through sequence diagrams extended with aspectual interception of messages.

GENERAL RULE: **createAspectAdvice(Aspect, Behavior, AdviceType)**

In the AMF framework

```
#operator: createAspectClass(AspectComponent, Behavior, AdviceType)
#pre: [not(definedClass(AspectComponent)), adviceType(AdviceType,´after´)]
#post: [definedClass(AspectComponent), definedMethod(AspectComponent, Behavior)]
#tasks:[
    @do: defineClass(AspectComponent, ´Object´),
    @do: implementInterface(AspectComponent, ´AspectIF´),
    @do: defineMethod(AspectComponent, ´postaction(Object):int´, ´General advice method´),
    @do: defineMethod(AspectComponent, Advice, ´This method should be called by postaction() ´),
    @do: warning(´Remember to update the aspect bank in AspectModerator with this aspect´)]
#end-operator.
```

framework-related planning operators

In the AspectJ language

```
#operator: createAspect(AspectComponent, Behavior, AdviceType)
#pre: [not(existingAspect(AspectComponent)), adviceType(AdviceType,´after´), adviceOption(´static´)]
#post: [definedAspect(AspectComponent), hasMethod(AspectComponent, Behavior)]
#tasks:[
    @do: defineAspect(AspectComponent),
    @do: defineAdvice(AspectComponent, ´static after():XXXName´, ´General advice construct´),
    @do: defineMethod(AspectComponent, Behavior, ´This method should be called by some advice´),
    @do: warning(´Remember to include advice XXXName in some poincut of AspectComponent´)]
#end-operator.
```

aspect-language-related planning operators

Fig. 8 Mapping of aspect rules to programming tasks for different aspect technologies

5.2 From the UCPOP to the SHOP2 planning algorithms

Smartweaver is a generalization of the *Smartbooks* documentation method, where besides framework documentation, we also have mapping rules to translate aspect modeling concepts into framework-specific counterparts. As an example of the aspect rules included in *Smartweaver*, Fig. 8 shows the functionality of creating an aspect for both the *AspectModerator* framework and the *AspectJ* language. Note how the high-level design task *createAspectAdvice()* can be implemented with a different set of programming tasks, according to the available technologies. Once a specific aspect technology is chosen by the developer, the planner is oblivious of the actual rule mapping.

In terms of planning, the goals in *Smartweaver* refer to high-level directives for aspect weaving. As illustrated by the format of operators of Fig. 8, the tool can call a UCPOP planner to accomplish the goals. A planning situation with UCPOP for the *AspectModerator* framework was actually exemplified in Fig. 4. Following the conventional UCPOP search, general aspect operators are progressively linked to more specific operators through precondition establishment. This way, the operators of the planning domain continue to be scripts, specified by the documenter either in a textual form or graphically. The first version of *Smartweaver* applied this planning schema to support the *AspectModerator* framework and a subset of the *AspectJ* language with success.

Although the basic planning algorithm remains the same than in *Smartbooks*, the introduction of aspects led to two modifications. First, a list of predetermined goals for the aspect domain was identified. Examples of these goals are: aspect creation, provision of advice before/after some method, association of an aspect with a functional component, declaration of precedence between aspects, etc. These goals can be selected by the developer or directly inferred from the *ASideML* diagrams. Additionally, new programming tasks were defined to handle the different language constructs of *AspectJ* [4], analogously to the tasks used for framework instantiation. Overall, the planning domain was updated with general aspect operators, framework-related operators and language-related operators. Before starting the planning algorithm, the user chooses the target aspect technology on which the output plan should be based on, and then the tool selectively enables different groups of operators and consistency rules for the planner. This switch of technologies shows the flexibility of having a general-purpose planner.

However, as we run some experiments with *Smartweaver*, the UCPOP planner exposed some limitations. The need of supporting various aspect technologies and evolving their capabilities caused a considerable overhead when writing the planning domain. Also, we observed that the concep-

Table 2 Comparison between the versions of UCPOP and SHOP2 for *Smartweaver*

Issue	UCPOP	SHOP2
Domain theory	√ The operators can be graphically specified by means of recipes in a UML-like notation. × The number of operators can be really large. ± It is capable of representing rules for frameworks/languages, although imposing a backward linking of operators. When specifying the planning domain, the documenter must be aware of this style of writing.	× The operators do not have yet graphical counterparts. Therefore, the scripts may be demanding to write and require knowledge of both the planner and the domain. √ It is possible to organize the operators in a hierarchical manner, in order to reduce the number of operators. ± Much of the planning problem seems to be really solved by the documenter writing the methods and operators, while the planner is relegated to the recovery of parts of the plan as prescribed by the specification.
World state and goals	√ The world state comes from the class diagrams opened within the project. ± The goals are selected by the user as aspect functionality items.	√ The world state comes from the class diagrams opened within the project. √ The goals are selected by the user as aspect functionality items. These goals can also present a hierarchical structure.
Support for design constraints	± Handling consistency among design representations may be tricky, even when the planner does not change the world state.	× As the world state may change while planning, consistency among design representations needs to be carefully handled.
Flexibility of the whole design	± The planner is able to re-plan when some goals change, or even plan only for a subset of the goals. However, the flexibility of the planning framework to support new features is limited.	√ Hierarchical task networks enable a flexible support for configuring different aspect technologies. × It cannot plan for subsets of goals. √ Adding new features as built-in planning routines is straightforward.
Understandability of problem-solving strategy	× Composite operators have to be expressed in terms of preconditions that enable more detailed operators, which often makes the specification awkward. ± The developer may get lost when dealing with user-query tasks, because he/she has no clue of the purpose of those tasks or their context. × As the planner goes backward, the developer cannot reason about where the planner was going to when satisfying a goal.	√ The user can see more clearly the direction in which the planner is searching. × User-query tasks may appear several times due to backtracking. √ The debugging process gets facilitated.
Performance	√ The backtracking is efficient. √ The planning process never changes the world state while constructing the plan.	√ The backtracking is efficient, if the planner is equipped with sufficient design knowledge to prune invalid options. ± The planner plans for tasks in the same order they will be later executed. This is desirable to speed the planning process, but often produces changes in the world state.

tual division between general and technology-specific operators suggested a hierarchical organization for the domain. Therefore, the UCPOP algorithm was replaced by the SHOP2 algorithm, which is nowadays the default planner for *Smartweaver*. The working of the planning engine with SHOP2 is similar to what was presented in Sect. 4.2, except for the absence of plan merging.

5.3 Lessons learned

The *Smartweaver* tool has been tested in several case-studies, featuring different aspect technologies such as: *AspectJ*, *AspectModerator* framework and *AspectWerkz*. The *AspectWerkz* framework is a blend between *AspectModerator* and *AspectJ*, in which aspects are incorporated to objects through proxies but using an AspectJ-like pointcut language specified in XML. Table 2 summarizes the strengths and limitations of the UCPOP and SHOP2 variants used for the experiments. Using either UCOPOP or SHOP2 as the underlying planning algorithm, developers concentrate on modeling his/her aspect-oriented application, and they are only aware of the high-level decisions necessary for that application.

An interesting result of HTN planning is that it gave us flexibility to change the weaving mechanisms with little effort. Although most of the analyzed aspect technologies implement a common set of features (e.g., advice, pointcuts, aspects, etc.), it may happen that some properties specified in the *AsideML* models have no translation to certain technologies. This may vary if the technologies provide dynamic or static support for aspects, as for example the *AspectModerator* framework and *AspectJ* respectively. Con-

versely, even using a particular aspect technology, we may have many alternative ways of implementing the same aspect functionality. Also, as the aspect paradigm evolves, new features and constructs may be added to the corresponding technologies. In all these situations, the documenter can take advantage of the HTN formalism to (re-)codify proper operators and points of interaction with the developer, so as to incorporate the new mechanisms into the *Smartweaver* tool.

There are still some technical issues that remain open in *Smartweaver*, namely: policies for determining the weaving order of aspects, policies for solving possible aspect conflicts, and definition of reusable aspect libraries. We think these issues cannot be treated through standard planning, but rather they should be encapsulated as utility modules to be called by the planner. Here, *JavaLog* has good credits to help us to implement such a "hybrid" planning framework. On the other hand, some knowledge-related problems discussed for the UCPOP planner were "inherited" by the SHOP2 planner too. For instance, the implementation of recurring aspect solutions within the same application cannot be always detected by the planner, loosing some opportunities to make the weaving process simpler. This problem appeared in *Smartbooks*, although it was partially solved by adding more instantiation rules.

6 Related work

Several researchers have recognized the value of AI techniques in assisting design activities, although most of the tools have been historically based on rule-based systems. Representative examples of rule-based systems for design are: the Programmer's Apprentice project, the critiquing mechanisms of Argo for object-oriented design, and the ArchE tool for architectural design. Nonetheless, we can find some interesting applications of planning to the areas of project management, dynamic configuration management and network security.

A rule-based system represents knowledge in terms of a group of rules that tell the system what it should do or what it could conclude in different situations [14]. The first experiments with design activities can be traced to the Programmer's Apprentice project at MIT to build an environment with automated support for requirements, design and implementation. This assistant, rather than being passive, would act as an autonomous agent able to cooperate with the engineer or take over aspects of various software-engineering tasks. With this purpose, the authors defined a kind of patterns called *cliches*, in order to capture design concepts and implementations techniques. Unfortunately, much of that work failed due to the weak support given by the representation formalism and the inherent complexity associated with the generation of operational programs from requirements. More recently, in the ADEF project [26], the authors have proposed a framework for eliciting architectural decisions from requirements. By means of a built-in mechanism of rules and decision trees, the framework maps some architectural properties to design elements. Along this same line of research, the Predictable Architecture Design (PAD) approach at the Software Engineering Institute [6] has focused on designing a software architecture in such a way it can predictably achieve its quality-attribute requirements. In fact, the PAD framework is the base of the architectural design theory used by *Designbots*. PAD is supported also by a rule-based system integrated with the Eclipse platform. In general, rule-based expert systems are a convenient platform for prototyping, as long as we keep the decision mechanisms relatively simple. However, rules have been regarded as inadequate to model complex decision mechanisms.

An interesting shift towards more active tools for design is proposed by the Argo environment [35]. The novelty of Argo is a "system of automated critics" integrated with traditional design functionality. These critics are a sort of agents that operate in background, and continually monitor the state of the design and the actions taken by the architect as the software architecture is being constructed. Each critic has a partial perspective (or view) of the architectural representation. When specific conditions hold, a subset of critics may get activated to give suggestions to the architects about the implications of, or alternatives to, a given design decision. Furthermore, the architect can respond to critics with some feedback, in order to make the interaction more effective. Critics are implemented as rules that consider also mixed-initiative. A limitation of critics is that, in spite of giving useful suggestions, they are not designed to actively manipulate design artifacts as in *Designbots* or to execute programming tasks as in *Smartbooks*.

The problem of framework instantiation has been tackled by Hakala et al. [22] within the Fred project. Basically, Fred is a research prototype that considers the instantiation interface of a framework as a set of "specialization patterns", and then provides interactive task-based programming assistance to use the framework. This view of patterns is closer to that of instantiation rules in *Smartbooks*. However, unlike *Smartbooks*, specialization patterns are not enacted through planning. A special algorithm based on constraint satisfaction techniques controls which patterns should be selected and which tasks should be executed by the developer. The Fred tool has been applied both to industrial frameworks or to reuse architectural standards such as Java Beans. Indeed, many of the ideas developed for this approach could be an interesting complement for *Smartbooks* and *Smartweaver*.

In [3], the authors propose the use of temporal planning to reconfigure distribute systems where plans are derived from a domain of basic architectural elements and operations. Once the initial configuration state and the constraints have been defined, there are goals that specify the required

states the system must go into, so that the planner can search for a suitable solution. The state, goals and operators are all expressed by means of scripts, where the programmer is responsible for codifying a number of situated rules with configuration changes. This schema works quite well as long as the configurability situations can be anticipated by the developer. However, it is not appropriate when the combination of actions and their possible interactions are unknown. Although the approach is still under experimentation, the planning results concur with our observations about domain writing and scalability of the planner to practical cases.

Regarding project management, CaBMA [29] is a tool that combines HTN planning and case-based reasoning to reuse pieces of project plans and assist the user in the creation of new projects. In this context, a case permits to capture developer's experiences in terms of the various elements of a project plan. This tool works integrated with MS Project^{TM}, supporting activities like: capture of pieces of plans in a base of cases, application of these cases to refine existent projects a even generate new ones, and consistency between cases. Two specialized algorithms are provided: one for collecting cases, and another one for applying these cases during project edition. The first algorithm identifies candidate cases and stores them in a generalized form. The second algorithm is an adaptation of SHOP2 in which tasks may be decomposed either by normal HTN methods or by generalized cases. Although CaBMA can be seen as an assistant, in the sense that it helps the user to define the structure of his/her projects, the approach does not take into account feedback of the results of its recommendations. Another difference with the tools reported in this paper, is that the project plans generated with CaBMA are not really executed on the world state.

7 Conclusions

In this article, we have discussed three types of software design, each of which has its own characteristics but also bears specific problems for the developer. In particular, we analyzed three problems with different abstraction levels, namely: framework instantiation (object-oriented implementation), composition of architectural styles (architectural design), and aspect weaving (detailed design). Although there are many techniques, abstractions and technologies available for solving these design problems, it is crucial for the developer to determine when and how to apply the right solution as well as to evaluate if the consequences of his/her decisions are the expected ones. This process can be seen as a "trial-and-error search" involving several sources of design knowledge. Therefore, we decided to investigate the utility of planning-based tools to assist the developer's activities.

As reported in the case-studies, we have applied two kinds of planning algorithms: UCPOP and SHOP2, which were adapted to fit the characteristics of the different "design domains". In general, automated planning proved to be very useful to help the developer to build a "design plan", based on the selection and articulation of a collection of "design operations" for each "design domain". However, we observed that the theory supporting these operations has a different interpretation, depending on the design abstraction and the artifacts involved. For example, in *Smartbooks* and *Smartweaver,* the final applications must be built according to the prescriptions of base frameworks, while in *Designbots* the resulting designs must satisfy certain quality-attribute values. This means that planning provides the "generative mechanism of actions" [37] for detailed design and implementation, mainly because the framework-related knowledge offers a restricted set of building blocks to construct the applications. In the second case, planning gives rather the "reasoning about action" [37] as it intends to capture an open-ended collection of architectural strategies and patterns, which eventually lead to an architecture with desired quality-attribute properties. Certainly, the latter conception of planning presents more complexity for tool support. In addition, a mixed-initiative schema complements the planning process for those decisions that cannot be made by the planning engine. Overall, the main contribution of planning-based tools targeted to design is that the developer gets focused on the goals and let the planner reason about the actions to achieve these goals, while he/she still keeps control of the leading design decisions.

One of the most critical issues for the effectiveness of the planner's decision-making during assistance is the search-control strategy to use. On one side, we can apply a regressive search (as in the case of UCPOP), and rely on a function that non-deterministically chooses a general operator and tries to establish preconditions for it. Although the UCPOP-based tools worked with a reasonable performance, there may be complex design tasks in which a backward strategy can complicate the developer's reasoning about the steps being carried out by the planner. The inability of UCPOP to specify operators in a hierarchical way is also a problem. On the other side, we can apply a forward search (as in the case of SHOP2), and rely on corresponding functions that choose operators/methods for a given task. This is more natural for the developer when dealing with design situations where the goals cannot be clearly specified, such as the quality-attribute goals to be accomplished by the *designbots*. However, the drawbacks come here from planning for tasks in the same order that they will be executed, which often add overhead when writing the domain specifications. Despite our collection of tactics, patterns and mechanisms for performance and modifiability were handled fairly well, we foresee that the specification of the HTN domain can be a

time-consuming and error-prone activity. In any case, therefore, the tool developer must evaluate the characteristics of the domain, and then make tradeoffs between efficiency and expressiveness of the planning language. Along this line, we are currently developing an HTN authoring tool to manage different sources of knowledge and to trace different planning executions. We are also analyzing how to improve the planning model to deal with resources in the operators.

Finally, the development of "intelligent agents" integrated with conventional CASE tools holds promising benefits for software design. In general, these agents take into account interests and preferences of the developers in order to provide guidance on how specific design tasks should be carried out. Furthermore, these agents can also execute routine tasks on behalf of the developer and directly manipulate parts of the design. Based on the results of the *Designbots* and *Smartweaver* tools, we can say that a planning approach is particularly useful for these agents, because it provides a flexible reasoning framework for the "constructive" process of design [31]. An interesting approach for improving decision-making in practical design situations is to complement planning with other AI techniques (e.g., case-base reasoning, Bayesian networks). On the other hand, gathering empirical data about human-computer-interaction aspects of planning-based tools is a topic of ongoing research.

Acknowledgements The authors would like to thank all the members of the ISISTAN Research Institute (UNICEN University) and practitioners that were involved in the development and usage of the planning tools. Also, the authors are grateful to the anonymous reviewers for their valuable comments on this manuscript.

References

1. Amandi A, Campo M, Zunino A (2004) JavaLog: a framework-based integration of Java and Prolog for agent-oriented programming. In: Ledley RS (ed) Computer languages, systems and structures. Elsevier Science, Amsterdam. ISSN: 0096-0551
2. AOSD homepage. Tools for Developers. http://www.aosd.net/wiki/index.php?title=Tools_for_Developers
3. Arshad N, Heimbigner D, Wolf A (2003) Deployment and dynamic reconfiguration planning for distributed software systems. In: Proceedings ICTAI 2003: pp 39–46
4. AspectJ homepage. http://www.eclipse.org/aspectj/
5. AspectWerkz homepage. http://aspectwerkz.codehaus.org/
6. Bachmann F, Bass L, Klein M, Shelton C (2005) Designing software architectures to achieve quality attribute requirements. In: IEE Proceedings on Software, August 2005, pp 153–165
7. Bass L, Clement P, Kazman R (2003) Software architecture in practice, 2nd edn. Addison–Wesley, Reading
8. Blum A, Furst M (1997) Fast planning through planning graph analysis. Artif Intell 90:281–300
9. Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal John M (1996) Pattern-oriented software architecture. A system of patterns. Wiley, New York
10. Campo M, Price T (1999) Luthier—-building framework visualization tools. In: Fayad M, Johnson R (eds) Implementing object-oriented application frameworks, frameworks at work. Wiley, New York
11. Campo M, Díaz Pace A (2001) Analyzing the role of aspects in software design. Commun ACM, special issue on Aspect-oriented programming 44(10):66–73
12. Campo M, Díaz Pace A, Zito M (2002) Developing object-oriented enterprise quality frameworks using proto-frameworks. In: Software: Practice and Experience, vol. 32. Wiley, New York, pp 837–843
13. Campo M, Díaz Pace A, Trilnik F (2004) "Computer, please, tell me what I have to do . . ." An approach to agent-aided application composition. Int J Soft Syst 74:55–64
14. Cawsey A (1997) Essence of artificial intelligence. Prentice-Hall, New York
15. Chung L, Nixon B, Yu E (1995) Using non-functional requirements to systematically select among alternatives in architectural design. In: Proceedings 1st international workshop on architectures for software systems, Seattle, 14–28 April 1995, pp 31–43
16. Constantinides C, Bader A, Elrad T, Netinant P, Fayad M (2000) Designing an aspect-oriented framework in an object-oriented environment. ACM Comput Surv 32:41
17. Díaz Pace A (2004) A planning-based approach for the exploration of quality-driven design alternatives in software architectures. PhD thesis, Faculty of Sciences, UNICEN University, Tandil, Argentina, September 2004
18. Díaz Pace A, Campo M, Trilnik F (2004) A knowledge-driven approach for aspect composition. In: Aspect-oriented software development. Addison–Wesley, Reading. ISBN 0-321-6-7
19. Elrad T, Filman R, Bader A (2001) Aspect-oriented programming—introduction. Commun ACM 44(10):29–32
20. Fayad M, Schmidt D, Johnson R (1999) Building application frameworks: object-oriented foundations of framework design. Wiley, New York
21. Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns, elements of reusable object-oriented software. Addison–Wesley, Reading
22. Hakala M, Hautamäki J, Koskimies K, Paakki J, Viljamaa A, Viljamaa J (2001) Generating application development environments for Java frameworks. In: Proceedings of the 3rd international conference on generative and component-based software engineering (GCSE'01), Erfurt, Germany, September 2001. Lecture notes in computer science vol 2186. Springer, Berlin pp 163–176
23. Johnson R (1992) Documenting frameworks using patterns. In: Proceedings OOPSLA 1992. Conference on object oriented programming systems languages and applications, Vancouver, BC, Canada. ACM Press, New York, pp 63–76. ISBN:0-201-53372-3
24. Kazman R, Bass L (1994) Toward deriving software architectures from quality attributes. Technical Report CMU/SEI-94-TR-10
25. Kiczales G, Lamping J, Mendhekar J, Maeda C, Videira Lopes C, Loingtier J, Irwin J Aspect-oriented programming. (1997) In: Proceedings of the European conference on object-oriented programming (ECOOP), Finland, June 1997. Lecture notes in computer science, vol. 1241. Springer, Berlin
26. Liu W, Easterbrook S (2003) Eliciting architectural decisions from requirements using a rule-based framework. In: Proceedings of the 2nd international workshop from software requirements to architectures (STRAW'03), co-location with ICSE 2003, Portland, OR, USA, May 2003
27. Long D, Fox M (2002) Progress in AI planning research and applications. In: Upgrade /Novatica 159. III (5), pp 10–25
28. Maes P (1994) Agents that reduce work and information overload. Commun ACM 37(7):31–40
29. Xu K, Muñoz-Avila H (2004) CaBMA: case-based project management assistant. In: Proceedings of the sixteenth innovative applications of artificial intelligence conference (IAAI-04). AAAI Press, Menlo Park
30. Nau D, Au T-C, Ilghami O, Kuter U, Murdock JW, Wu D, Yaman F (2003) SHOP2: an HTN planning system. J Artif Intel Res 20:379–404

31. Nau D, Ghallab M, Traverso P (2004) Automated planning: theory & practice. Kaufmann, San Francisco. ISBN:1558608567
32. Ortigosa A, Campo M, Moriyón R (2000) Towards agent-oriented assistance for framework instantiation. In: OOPSLA 2000 (Conference on object-oriented programming, systems, languages, and applications), October 2000, ACM SIGPLAN, vol 35, pp 253–263
33. Ortigosa A, Campo M (1999) Smartbooks: a step beyond active cookbooks to aid in framework instantiation. In: Technology of object-oriented languages and systems, TOOLS'99. IEEE Press, New York, July 1999
34. Ortigosa A, Marcelo M (1999) Using incremental planning to foster application frameworks reuse. In: International conference on software engineering & knowledge engineering. SEKE'99, 16–19 June 1999, pp 362–369
35. Robbins J, Hilbert D, Redmiles D (1996) Extending design environments to software architecture design. Int J Autom Softw Eng, special issue: The Best of KBSE '96
36. Shaw M, Garlan D (1996) Software architecture, perspectives on an emerging discipline. Prentice-Hall, New York
37. Smith G, Gero J (2005) What does an agent mean by being "situated"? In: Design studies, vol 26, pp 535–561
38. Tate A, Dalton J, Levine J (2000) O-plan: a web-based AI planning agent. In: AAAI-2000 intelligent systems demonstrator. Proceedings of the national conference of the American association of AI (AAAI-2000), Austin, TX, USA, July 2000
39. Tekinerdogan B (2000) Synthesis-based software architecture design. PhD dissertation, University of Twente, Enschede, The Netherlands. ISBN 90-365-1430-4
40. von Flach Garcia Chavez C (2004) A model-based approach to aspect-oriented design. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, PUC/RJ, Brasil
41. Weld D (1994) An introduction to least commitment planning. AI Mag 15(14):27–61
42. Wilkins D, desJardins M (2001) A call for knowledge-based planning. AI Mag 22(1):99–115
43. Yang Q (1997) Intelligent planning: a decomposition and abstraction based approach. Springer, New York