Collective Intelligence: An Agent-Based Approach to Programmatic Organization in Architecture

Yaniv (Junno) Ophir

Collective Intelligence: An Agent-Based Approach to Programmatic Organization in Architecture

Yaniv (Junno) Ophir

Architectural programming is the research and decision-making process that identifies the scope of work to be designed. Programming is difficult because it involves identifying, collecting, analyzing and updating information from different sources such as engineers, clients, users, consultants, and others. In this paper I propose a computational model for programming and describe its implementation, a tool called PENA that allows a programming expert to represent different processes and people involved in a project using intelligent agents. By delegating responsibility to agents, a programming expert can better organize and manage project data as well as find creative solutions to conflicting issues through agent negotiation. As a proof-of-concept, I show how an agent, called the Arch-Learner, manages adjacencies of rooms in a simple program for a house by clustering them into public and private rooms. I conclude with a discussion of future work and development of PENA.

1. Introduction

Peña and Parshall [1] describe architectural programming as the task of assessing the various complexities of a design project through multiple perspectives. The goal of programming in architecture is to be able to translate all the collected information of a project into an initial floor plan layout and information model. These will then be the starting point of the project's next phase, i.e., the design of its architectural form. One of the major challenges of programming is avoiding the *data clog* by having the ability to organize and manage enormous amounts of data from different sources in a way that exposes a project's potential. This is why architectural programming is sometimes referred to as the *problem seeking* stage, where the formal design of the project is called the *problem solving* stage. Some of the benefits of a successful programming process are: avoiding delays and saving money by identifying problems in advance, exploring different design options, meeting client-user needs, and so on.

Within architectural programming, the automated design of a floor plan layout has captured the attention of many researchers from various disciplines. Mitchell [2], for example, discusses the merits of automated design in architecture and also offers a method for generating least cost floor plan layouts [3]. In the past few decades various computational tools have been developed for automated design of floor plan layouts [4-5, 8-18]. In general, these tools attempted to find the best possible floor plan layout to a given objective and set of constraints by searching through the space of all possible solutions. However, I believe that their approach does not begin to answer the complexity of architectural programming where there is emphasis on consolidating dynamic data to produce a range of possibilities rather than any specific solution, be it optimal or not. Furthermore, finding optimal floor plan layouts via an automated process requires that the user specify exact constraints and objectives which are seldom fixed, explicit or enumerable in any architectural project [6].

It seems that the increasing amount and complexity of information in today's building industry requires a change in how a programming expert, for example, might interact with that information. Now, more than ever, those working with information models such as Building Information Modeling (BIM) could use an extra pair of hands as well as eyes. I believe that autonomous and intelligent agents can play the role of a designer's proxy by assuming responsibility for certain parts of a project and making decisions regarding those parts on behalf of the programming expert.

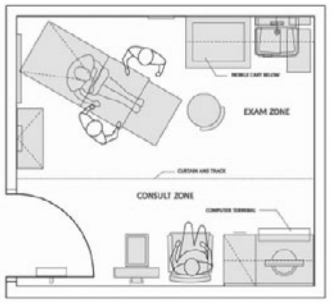
In light of the aforementioned, I propose a computational model for architectural programming called PENA – Programmatic Planning Environment for Negotiating Agents. PENA assists the programming expert by representing the various processes and people involved in the project as autonomous intelligent agents. For example, PENA might have a sustainability agent that manages information about the project's energy consumption and other related issues. Another agent, the budget agent, might keep track of project spending and, if needed, communicate with the sustainability agent about the cost of its current energy rating. By encapsulating the different perspectives of a project within agents, a programming expert can achieve greater control over his domain and ultimately become more efficient and creative.

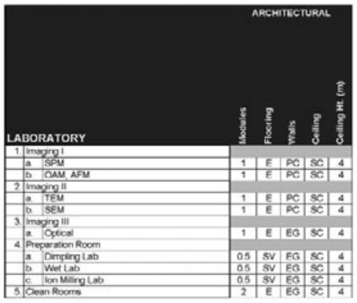
This paper is structured such that in the following section I provide an overview of architectural programming by looking at a specific example from the literature and its application in practice. Then in section 3, I take a quick look at one key aspect of architectural programming, i.e., the automated design of a floor plan layout, through major precedents. In section 4, I briefly present the characteristics of multi-agent systems and hint to a potential integration with BIM. Finally, in section 5, I describe the computational model and its implementation, namely PENA and in section 6, I show how an agent can manage room adjacencies in a simple program for a house. Then, because PENA is still a work-in-progress, I offer some interim conclusions and ideas for future work.

2. Programming in practice

The field of architectural programming was first formalized by Peña and Parshall [1], who defined programming as: establishing goals, collecting and analyzing facts, uncovering and testing concepts, determining needs and stating the design problem. Furthermore, Peña and Parshall characterized design as a form of *problem solving* while programming was a process focused on *problem seeking*. With this distinction in mind, Peña and Parshall derived a unique definition for creativity in programming which depended on how the various parts identified during programming were organized. Peña and Parshall's programming methodology has influenced many architectural practices over the years, one of which is Payette [7]. I have been working with Payette's programming department in order to better understand how Peña and Parshall's methodology manifests in real-world projects. While I realize that Payette is only one example of many, I believe that the work of its programming department is representative of what is currently happening in the field. Furthermore, Payette's programming department has a well structured programming process that is ideal for capturing in a computational model such as the one I present in this paper. Therefore, I would like to briefly discuss the practical implementation of Peña and Parshall's programming methodology as it is used at Payette.

The programming process at Payette begins in a meeting with the client's steering committee to establish the project's goals and objectives. These goals fall under many different categories such as operational goals, technical goals, image goals, budget goals, etc. Moreover, they are usually expressed by using simple, everyday concepts such as contemporary, high-tech, connected, exposed, and others. Each of these concepts maps to a set of constraints that must be understood by the client and tailored to his needs. In other words, if a client wants a building that exposes its structural system then he must realize that his decision imposes certain constraints on other aspects of the project such as privacy and sustainability. Programming continues with the collection and analysis of data through meetings with users, site surveys, personal questionnaires, precedents, standards, etc. Current tools at the programming expert's disposal to organize and manage the incoming data are spreadsheets, tables, diagrams and databases. As data begins to accumulate, its organization and management become crucial. In fact, if we recall Peña and Parshall's definition of creativity in programming, this stage in the process has the greatest influence over the project's final outcome. However, this is only part of a greater challenge facing the programming expert, who also has to cope with changes to the data further down the line and the way in which these changes might affect the project. In the programming jargon this problem is often referred to as the *data clog*. With the initial data collected and analyzed, programming proceeds to determine the project's needs. At this stage of the process, the programming expert utilizes space diagrams and tabulation (Figure 1) to calculate space requirements that respond to the data collected in the previous stage.

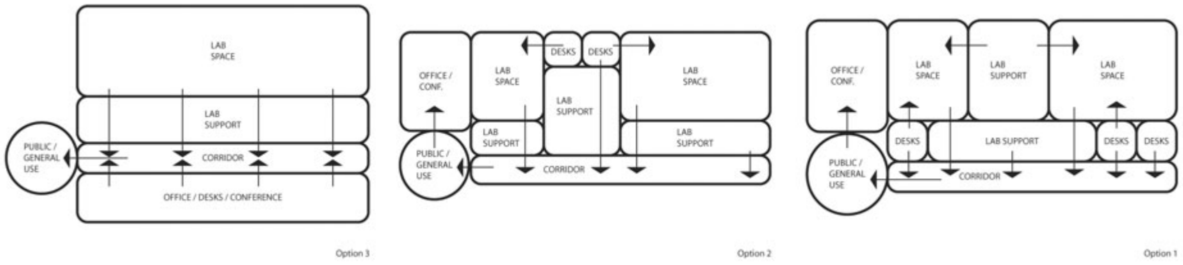




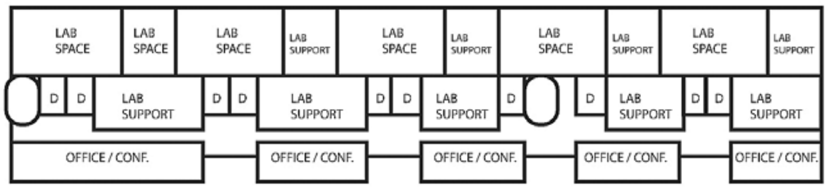
LABORATORY			ARCHITECTURAL				
			Modules	Flooring	Walls	Ceiling	Ceiling Ht. (m)
1	Imaging I						
	a	SPM	1	E	PC	SC	4
	b	OAM, AFM	1	E	PC	SC	4
2	Imaging II						
	a	TEM	1	E	PC	SC	4
	b	SEM	1	E	PC	SC	4
3	Imaging III						
	a	Optical	1	E	EG	SC	4
4	Preparation Room						
	a	Dimpling Lab	0.5	SV	EG	SC	4
	b	Wet Lab	0.5	SV	EG	SC	4
	c	Ion Milling Lab	0.5	SV	EG	SC	4
5	Clean Rooms		2	E	EG	SC	4

► Figure 1. Left: Space diagram. Right: Space tabulation [7].

The programmer may also make use of bubble diagrams to determine spatial affinities and adjacencies that begin to explore different spatial relationships between parts of the project. Informed by previous stages, programming continues with the development of concepts that further anchor the spatial organization of the proposed building. For example, the building's internal organization might be influenced by the social structure of its future users. In order to explore and present different organizational concepts, the programming expert uses diagrams such as the ones in Figure 2. When the client, together with the programming expert, has selected one of the spatial concepts, its diagram is further developed to represent more details (Figure 3). Finally, the programming phase concludes by extracting the key aspects that shape the project's successful completion. This is sort of an overview of what the building wants to be and will serve as the starting point for the next phase, the architectural design of form.



▲ Figure 2. Diagrammatic floor plan layouts [7].



◀ Figure 3. Detailed floor plan layout [7].

3. Automated design of floor plan layouts

In the previous section, I showed that the outcome of a programming phase is a database, or information model, of the project and its representation as a more or less schematic floor plan layout. Over the past few decades, a number of methods have been developed for the automated design of floor plan layouts (Table 1). Mitchell et al [8] developed a method for finding all topologically distinct arrangements of rooms within given constraints which satisfy a set of user-defined requirements. The method, also known as rectangular dissections, can also find the room dimensions by solving a set of linear equations and in some cases finds the dimensions for the least cost floor plan layout by linear programming. A different approach taken by Sharpe et al [9] experimented with the Metropolis algorithm and its application to the quadratic assignment layout problem, i.e., finding the optimal way of allocating a set of activities (A) over a set of zones (Z) within a series of time periods. Their method defines an optimal solution as the spatial configuration that produces the minimum cost generated by interactions between activities.

In more recent attempts to automate the design of floor plan layout, Keatruangkamala and Sinapiromasaran [17] developed an interactive program that uses Mixed Integer Programming (MIP) to find optimal solutions or layouts. The method enables the user to specify design constraints, such as fixed position and boundary, which in turn speed up the computation by pruning the space of possible solutions. Another approach is presented in Bier et al [18], where a search based on Boolean satisfiability (SAT) is carried out by a SAT solver. The solver, MiniSat+, converts a set of user defined constraints into Boolean clauses and then searches for a combination which satisfies these clauses. Before the solver can begin searching, however, the method has to get the boundaries of the design from the user and break them up into voxels, or volumetric pixels. Only then can the method begin to search for a spatial arrangement that will fit into the current voxel scheme. The method reduces the search space by applying heuristics such as identifying parts of the space that are difficult to access or too small to accommodate certain objects in the program.

While the methods described above can produce floor plan layouts comparable to those produced by a programming expert, I bring them up because they represent a very narrow and specialized approach to architectural programming, one which ignores crucial aspects of architectural design such as client-user demands, environmental impact, structural constraints and more. It seems to me that automation, even when it arrives at an optimal solution, does not help the designer, especially in the initial stages of the design, to better understand the complexity and potential of the problem. Automation and optimization in architectural design rely on the project's state at a given time. However, design is a continuous process where things change over time, and therefore, any computational process must be able to adapt to the changing circumstances of the design and to inform the designer accordingly. For instance, Combes says:

> "What will be more important for the architectural purpose will be to gain some knowledge of the overall range of possible classes of solution, and of how this total range of solutions is distributed over the dimensions of the solution space." [19]

Fortin adds:

> "Since the layout problems are usually ill-defined, one can ponder the usefulness of finding an optimal solution. A few suitable solutions are probably better." [20]

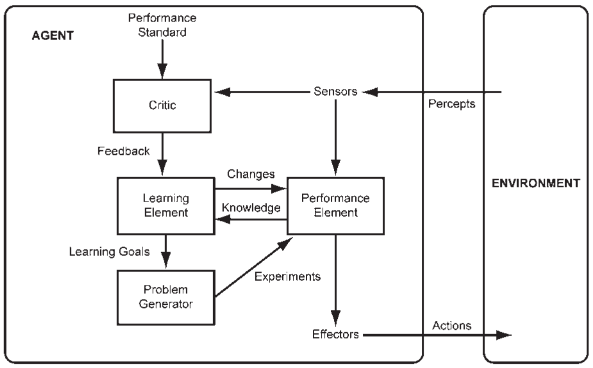
► Table 1. Methods for automated floor plan layout design.

Author	Year	Method
Mitchell et al [8]	1976	Constraint propagation, Non/linear programming
Sharpe et al [9]	1984	Simulated Annealing (Metropolis)
Flemming et al [10]	1992	Generate-and-test, Disjunctive Constraint Satisfaction
Schwarz et al [11]	1994	Branch-n-Bound
Jo, Gero [12]	1996	GA (Genetic Algorithm)
Medjdoub & Yannou [13]	1999	CSP (Constraint Satisfaction Problem)
Michalek [14]	2001	Various
Nath & Gero [15]	2004	SOAR, search, pattern matching
Arvin [16]	2004	Physical Dynamics
Keatruangkamala & Sinapiromasaran [17]	2005	Mixed Integer Programming (MIP)
Bier et al [18]	2008	SAT solver (miniSAT+)

4. Multi-agent systems in design

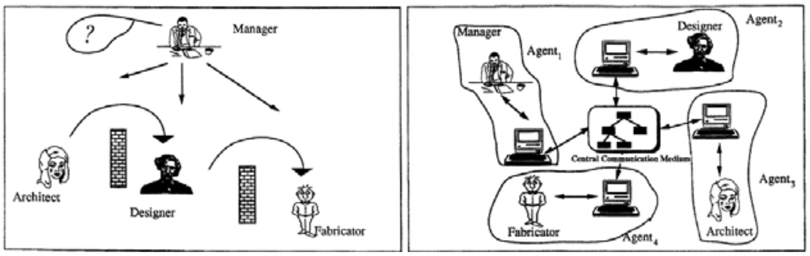
Multi-agent systems and specifically their use in design is a broad field that extends beyond the scope of this paper. However, in this section I will briefly touch on the following three points. First, I will give a definition of what are agents and multi-agent systems. Second, I will touch on Building Information Modeling (BIM) and the potential connection to multi-agent systems. Third, I will elaborate on the Designer Fabricator Interpreter (DFI), a system developed by Werkman [21-22], which inspired the model I propose in this paper.

An agent in computer science and artificial intelligence is defined as any entity that is capable of perceiving its environment and carrying out goal-directed action (Figure 4). Agents can be simple, like a thermostat that senses changes in the environment and adjusts the temperature accordingly, or complex, like the ones used in stock markets for trading purposes. When a system, like an online trading application, is composed of several agents interacting amongst themselves, it is usually referred to as a multi-agent system, sometimes known as a self-organizing system. In recent years, multi-agent systems have become somewhat of a trend mostly due to their ability to break a tough problem into several smaller, more manageable parts. Furthermore, multi-agent systems are considered flexible because they allow the addition, modification and reconstruction of agents without having to rewrite the whole system. Another aspect which makes multi-agent systems popular is their ability to function autonomously due to the agents' self-sufficiency and ability to communicate with their neighbors to resolve problems and accomplish goals.



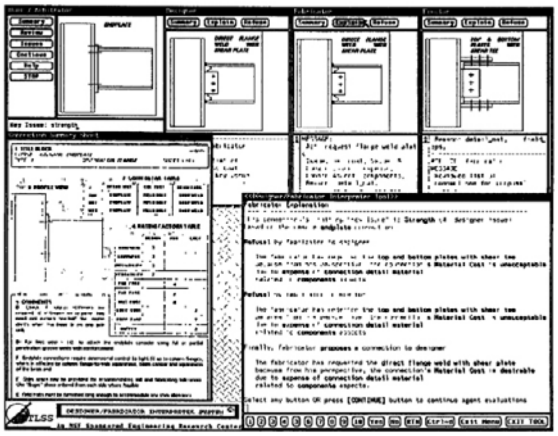
◀ Figure 4. A learning agent [28].

Sriram et al [23] point to the change that has taken place in the traditional product development process. The shift from over-the-wall engineering to a multi-agent view (Figure 5) has resulted in reduced development times, fewer changes and better overall quality. More importantly the use of multi-agent systems in product development has helped reinforce the collaborative nature of the process by providing a proxy for each of the parties involved. I believe that this proxy, or agent, is gradually becoming a necessity as the information involved in the different design domains increases in size and complexity. Let us take as an example the notion of Building Information Modeling (BIM). BIM is the process of generating and managing the data of a building during its life cycle [24]. The data encapsulated in BIM covers geometry, spatial relationships, geographic location, quantities and properties of building components. In some ways, one can think of programming as that stage in the design process when the project's BIM begins to take shape. As more and more information starts to come in and BIM's structure becomes intricate, the question of managing the information inside BIM in a productive way becomes crucial. This is where I believe agents and multi-agent environments present the greatest contribution. By enabling the designer, a programming expert in this case, to delegate responsibility for various managerial tasks, a multi-agent approach increases his ability to see the big picture instead of getting lost in the details.



► Figure 5. Left: Over-the-wall engineering. Right: Multi-agent view [23].

The multi-agent model I present in this paper is inspired by the Designer Fabricator Interpreter (DFI) [21-22], a knowledge-based tool which uses multiple agents to evaluate a design and comment on it from the agents' unique expert perspectives (Figure 6). DFI is used by structural design engineers in the preliminary stages of beam-to-column connections in buildings to find a solution that best meets the user's requirements. In a nutshell, DFI lets the designer pick an initial steel connection and key design issue. Then, control is passed to an arbitrator agent who selects one of three agents – Designer, Fabricator or Erector – to propose a connection that improves the key issue specified by the designer. At this point, connection proposals pass from agent to agent, each trying to improve the design from their unique expert perspective. For example, the fabricator agent may propose a connection that improves the designer's key issue and at the same time is easier to fabricate. When a disagreement arises between two agents, the arbitrator agent steps in and attempts to resolve the issue by either relaxing the importance of certain parameters in the current proposal or reverting to the last agreed upon proposal. If the arbitrator fails to resolve the conflict or a better solution cannot be found by any of the agents, then control is passed back to the user, who either accepts the current proposal or restarts the process.



```
<<Designer/Fabricator Interpreter Tool>>
Fabricator Explanation
[illegible] Strength [illegible] designer [illegible]
[illegible] endplate [illegible]

Refusal by fabricator to designer

   The [illegible] top and bottom plates with shear tee
   [illegible] Material Cost [illegible] unacceptable
   [illegible] expense of connection detail material
   related [illegible] components [illegible]

Refusal [illegible]

   The fabricator has [illegible] the top and bottom plates with shear tee
   [illegible] Material Cost [illegible] unacceptable
   [illegible] expense of connection detail material
   related to components [illegible]

Finally, fabricator proposes a connection to designer

   The fabricator has requested the direct flange weld with shear plate
   because from his perspective, the connection's Material Cost is desirable
   due to expense of connection detail material
   related to components aspects.

Select any button OR press [CONTINUE] button to continue agent evaluations
```

◀ Figure 6. The Designer Fabricator Interpreter [21-22].

5. PENA

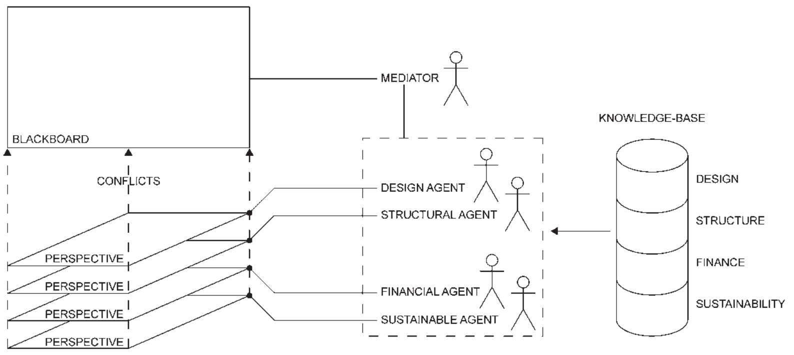
The Programmatic Planning Environment for Negotiating Agents, or PENA, is a tool based on a computational model that employs multiple agents to represent the various processes and people taking part in the architectural programming phase of a project. By delegating responsibility to autonomous intelligent agents, a programming expert can better organize and manage the information flowing in from different sources such as clients, users, consultants and others.

During architectural programming it is imperative that the programming expert have a clear view of the big picture. His job is to ultimately make sense of all the data collected and to be able to translate that data into design solutions. In practice, programming is a process that requires continuous negotiation and cooperation between the different people involved in the project. PENA allows the programming expert to represent the different and sometimes conflicting perspective of a project through the use of agents. These agents are simple computational entities that are able to sense the state of the project and perform actions on behalf of the programming expert. Agents in PENA also have the ability to create new agents themselves so as to produce a hierarchical structure that breaks down complicated tasks into smaller and more manageable ones. For example, if an agent that is managing adjacencies of rooms decided that two rooms cannot be adjacent for some reason, he could create an agent that took care of finding a new place in the floor plan layout for one of these rooms.

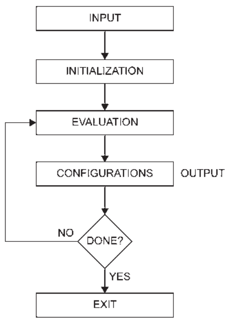
PENA is an interactive tool and relies on feedback from the user in order to instantiate the different agents that take part in the programming process. PENA is not intended for finding optimal solutions but is instead part of the distributed cooperative problem solving paradigm. I believe that PENA's approach reflects the distributed nature of the programming expert's job. It enables the programming expert to capture the decision making mechanism of different stakeholders involved in the programming phase such as those of the sustainability consultant, financial consultant, structural engineer, the user and the designer himself. The ability to separate the multi-tiered process of decision making enables the programming expert to better understand the cause-and-effect of certain decisions over the final outcome. Furthermore, through access to the agents' communication history, the programming expert can review the programming process and expose any potential problems in the design. Therefore, PENA gives rise to what I believe is the most important aspect of architectural programming, namely *problem seeking*.

5.1 System outline

An outline of PENA is shown in Figure 1 and a flow diagram in Figure 2:

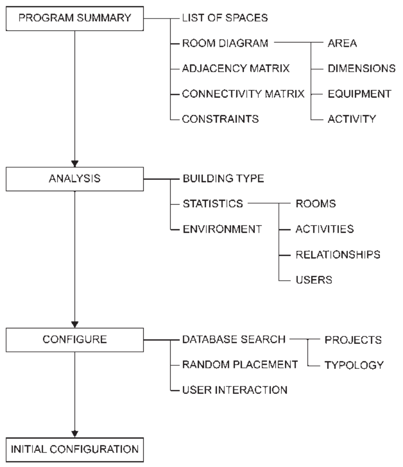


► Figure 7. PENA system outline.



► Figure 8. PENA system flow.

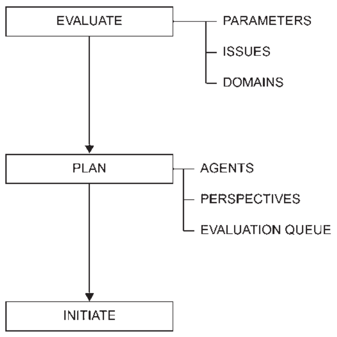
PENA starts by loading the project's data, i.e., the program summary as compiled by the programming expert (Figure 9). By default, the program summary contains a list of all the rooms in the project as well as their dimensions. However, the program summary may also include an adjacency matrix, connectivity matrix and a list of constraints. When the data is loaded, a matching procedure attempts to determine the project's building type, its major attributes and context by comparing it to previous projects it has stored in memory.



◀ Figure 9: Generating the initial configuration.

PENA can do this because it stores the semantic structure of previous programming projects it handled so if, for example, it is presented with a new problem, it can start decomposing the problem into semantic descriptions and comparing them to its existing knowledge until it finds a match. To use an example, let us say that the user has loaded data for the programming of a private residence. The project's program contains a list of rooms such as a kitchen, living room, bedroom, laundry, etc. PENA takes this input and turns it into a semantic representation. Having that layer of semantic meaning, PENA can then go on to recognize the project's building typology. In other words, if the semantic representation includes things like *has(thing, kitchen)*, *has(thing, laundry)* and *has(thing, living room)* then PENA can infer that *thing* is a house. The ability of any CAD software to understand the context of the problem the designer is working on is something that I believe does not yet exist and is absolutely necessary if we are to have computers that *understand* design. Once PENA has loaded the project data and inferred the context, control is passed to the user which allows him to either create the initial floor plan layout himself or have one created for him. If the user opts to design the initial floor plan layout himself he may use the *create room* menu to select one of the rooms from the program and place it on the PENA canvas. I discuss the *create room* menu and canvas later on in this section. When the user has placed a room on the canvas, he can then drag it around until he is satisfied with its position. The user goes on to create another room and then another until all the rooms are placed on the canvas. However, if the user decides to let PENA generate the initial configuration, the program will use examples stored in memory that fit the project's context to figure out how to arrange the given list of rooms into an initial floor plan layout. PENA will first try to group similar rooms together to create clusters that break down the problem into smaller and more manageable parts. So, for instance, if the program has a living room, dining room and kitchen, PENA will search the knowledge base it has constructed out of given examples and conclude that these three rooms form a cluster. When all rooms have been clustered, PENA will try to first organize all the clusters and then each cluster in itself. If PENA gets stuck or is done creating the initial floor plan layout, it will turn control over to the user so he may interact with the current solution.

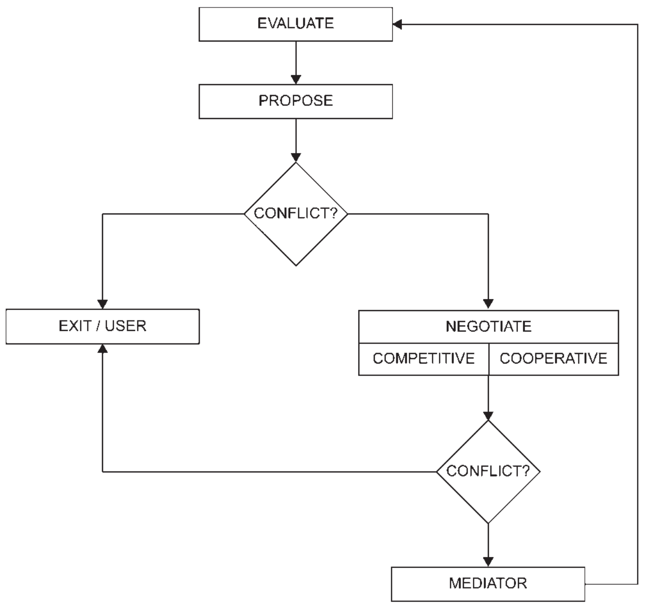
In the initialization stage (Figure 10), PENA takes the newly created initial configuration and does some pre-processing in order to get everything ready for the evaluation stage. The initialization stage includes an evaluation of the current configuration to initialize parameters, create design issues relevant to the project and establish domains which are the agents' perspectives. Next, the mediator agent that is responsible for the negotiation process creates a plan for the evaluation stage. The mediator first assigns agents to the different problem domains and then goes on to establish links between their different perspectives. Linking agents' perspectives is the basis for their negotiation abilities, e.g., it enables the sustainable agent to understand that changing the size of a window affects the cost of the room which in turn concerns the financial agent. In PENA these perspectives are implemented as a relational graph where agent issues are nodes and the connection between two issues is an edge. For example, cost and light are nodes that are connected to a window represented by another node and the window node is in turn connected to a dimension node. Any agent that attempts to change the dimensions of a window triggers a chain reaction that propagates up to the agents. The relational graph enables the indirect connection through issues amongst agents which is the foundation of their ability to understand each other's perspective.



► Figure 10. Initialization.

Once agents have been assigned and their perspectives linked, the mediator examines which agent is most affected by the state of the current configuration, that agent is placed at the head of the *evaluation queue*. The mediator goes on to order the remaining agents in the *evaluation queue* based on their relevance to the current configuration. For instance, if the initial configuration lacks windows then the environmental agent might be first in line to propose a change to the floor plan layout, followed by the designer agent and finally the finance agent. Once the evaluation begins the mediator will have to re-examine the agents' relevance to the current configuration and reorder the *evaluation queue* if needed. Finally, the mediator is ready to initiate the next phase.

The evaluation phase (Figure 11) starts with the first agent in the queue evaluating the current configuration. If the evaluating agent is the environmental agent, for example, then he might want to look at how much natural light each room is getting or if the rooms he knows are meant to have a certain level of lighting are actually getting what they need. Based on the results of the evaluation, the agent can propose to keep the current configuration or make changes to it based on his personal evaluation criteria. If the agent chooses to make changes to the current configuration, he sends his proposal to the mediator agent who runs a check for any conflicts between the current configuration and the agent's proposal. If there are no conflicts, the evaluation terminates and control is given back to the user. However, if a conflict arises, the mediator will determine which agents are most affected by the issue and order these agents to negotiate a solution. The negotiating agents can decide whether to work cooperatively or competitively based on the issue at hand. If the remaining agents arrive at a solution the evaluation process terminates and control is given back to the user. On the other hand, if the negotiating agents are in a deadlock, control is passed back to the mediator who attempts to resolve the situation and restart the evaluation process. To make sure the process terminates safely a counter keeps track of the evaluation cycles and triggers an *exit* command if the maximum number of cycles has been reached. Furthermore, the user has access to buttons on the interface of the agent manager menu that he can use to stop the agents.



◀ Figure 11. Evaluation.

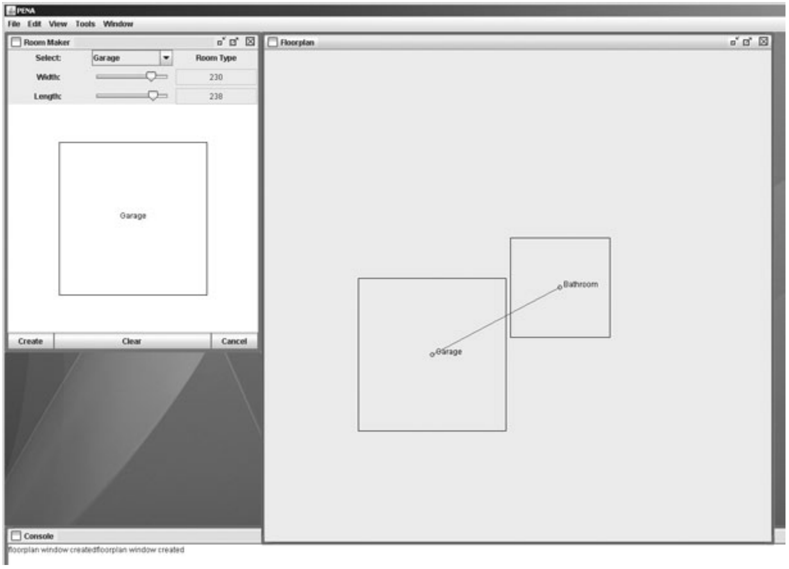
When the entire process terminates, control is given back to the user along with a floor plan layout and the history of the evaluation process. At this point the user can choose whether to exit the program completely or restart the evaluation process. If he decides to restart the evaluation, the user may choose whether he wants the current configuration to be the starting point of the new evaluation process or whether he would like to start from scratch.

5.2 PENA Components

In this section I present some of the components that make up PENA, namely the canvas, the different menus and agents. PENA's interface is implemented using the Java Swing Desktop component so it enables the user to open, move, maximize/minimize and close several windows simultaneously. I chose this interaction scheme because I believe any design process relies heavily on the designer's ability to multi-task. Having the ability to work in multiple windows facilitates this need for multi-tasking.

The PENA canvas (Figure 12) is the viewport through which the user interacts with the evolving floor plan layout. The canvas can take on two separate representations, the topology view and floor plan layout view. The topology view is a kind of enhanced bubble diagram which is implemented as a graph to represent all the objects in the scene and the relationships amongst them. I say the bubble diagram is enhanced because the topology view specifies a unit's area, location, and subordinate components, as well as its relation to its attributes such as materials, openings, etc. The second part of the canvas is the floor plan layout view which shows the current state of the floor plan. The floor plan layout view is in fact the geometrical representation of the topology view and echoes any changes the user makes in that view and vice versa.

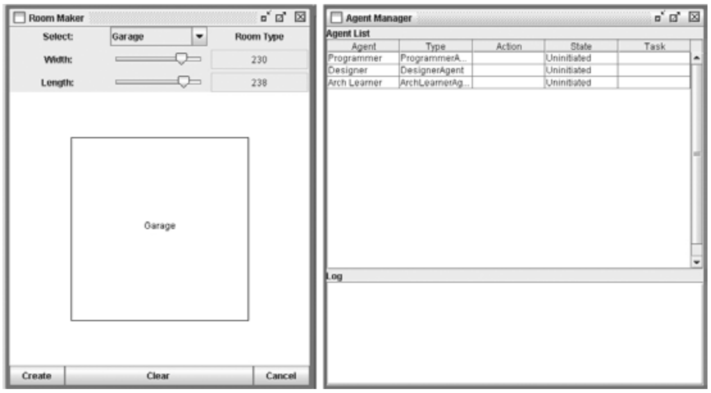
► Figure 12. PENA canvas.



```
floorplan window createdfloorplan window created
```

PENA has several menus to extend its functionality and interaction with the user. The *create room* menu (Figure 13) enables the user to select a room from the program, manipulate its dimensions and add it to the floor plan layout. The *agent manager* menu (Figure 13) opens up a table-like window through which the user can get information about the agents that are currently in use as well as interact with their properties, e.g., stopping, suspending and resuming their processing capabilities. Finally, the user can open the *train agent* menu which he can use to teach any agent "new tricks." In the following section, you will see how this menu was used to teach an agent that private rooms should not be adjacent to public ones.

The agents in PENA provide the core capabilities for distributed problem solving and encapsulate the expert knowledge needed in order to evaluate a proposed floor plan layout. Each agent in the system has at least the following elements: decision variables, unique perspective and communication mechanism.



Agent	Type	Action	State	Task
Programmer	ProgrammerA...		Uninitiated	
Designer	DesignerAgent		Uninitiated	
Arch Learner	ArchLearnerAg...		Uninitiated	

◀ Figure 13. Left: Create room menu. Right: Agent manager menu.

Decision variables make up the set of issues that are important to the agent. For example, location of openings is an important decision variable if you're an environmental agent and one of your concerns is the amount of natural light in each unit. Obviously, different agents have different decision variables which correspond to their unique perspective. Currently, these decision variables are implemented as a vector of objects of type issue. For example, connectedness, organization, and interaction are objects that inherent from the type issue and are stored in a unique vector that belongs to the designer agent.

An agent's perspective is a coupling between the agent's decision variables, i.e., his local view, and a shared objective function, i.e., the agents' global view. An agent's perspective is a powerful concept because it allows the creation of *interagent* relationships. For example, the size of windows can influence the amount of light in a room which is the environmental agent's concern but it can also affect the cost of constructing the room which concerns the financial agent. Through the agents' different perspectives an *interagent* relationship is created. As I mentioned previously this capability is implemented as a relational graph.

Each agent has a communication mechanism which enables him to query other agents for information, respond to queries from other agents or the user, negotiate about proposed floor plan layout and propose his own. Agents use a KQML-like protocol to construct their messages. KQML [25] seems to be a convenient way of describing an agent's intent through the several key points that all agents share. All communication between agents is saved in a text file for the purpose of tracking changes, discovering problematic issues and user inspection.

At this point I do not go into further details about the implementation of PENA. Readers are encouraged to see Werkman [21-22] for more information.

6. Case study

In this section I would like to give a very simple example of using PENA to create a floor plan layout that clusters a program for a house into private rooms and public rooms. Because PENA is still in the early stages of development, the purpose of this example is to show that one, PENA has the potential to become a flexible and intuitive tool for programming experts and two, PENA accomplishes this by taking a complicated task and breaking it up into several simpler tasks.

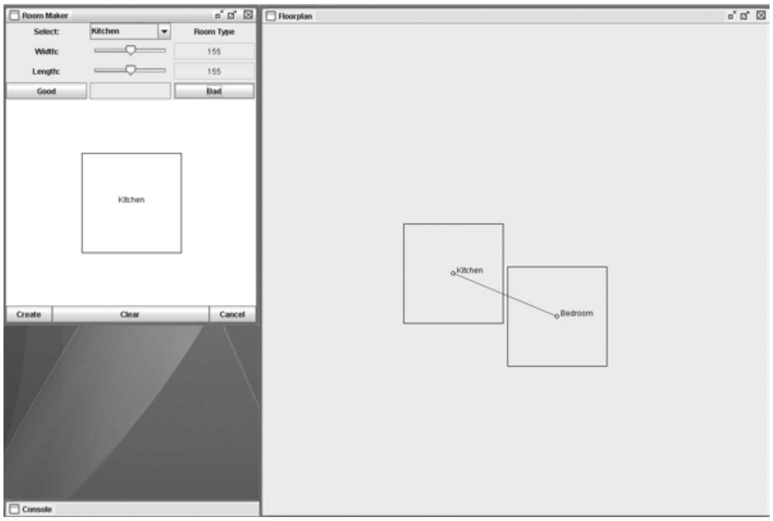
For this example, I used a simple program for a house (Table 2) to explore how PENA could help a programming expert produce a floor plan layout that clusters the program into private rooms and public rooms.

► Table 2. Simple program for a house.

Name	# of units	Dimensions (ft.)
Garage	1	22x22
Living room	1	13x19
Kitchen	1	12x10
Family room	1	12x12
Master Bedroom	1	13x12
Master Bathroom	1	10x8
Bedroom	2	10x10
Laundry/Utility room	1	6x8
Bathroom	1	6x8

Examining the precedents I covered in this paper and the working methodology of programming experts in practice, it seemed that any automated process attempting to organize rooms on a floor plan layout required the user to specify an adjacency matrix. Manually specifying adjacencies between rooms is a tedious and time consuming task that ultimately prevents the programming expert from focusing on the floor plan layout as a whole.

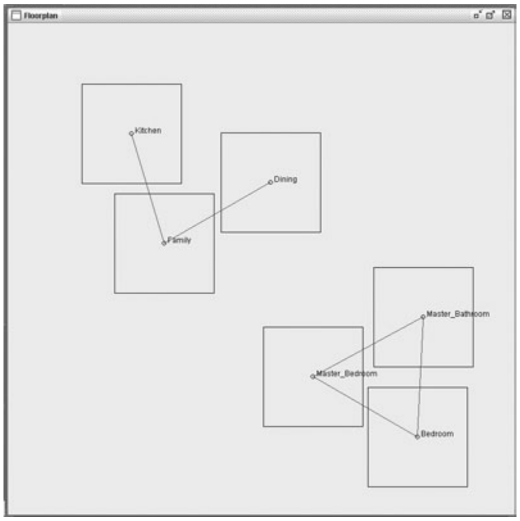
In order to address this issue, I implemented two agents in PENA, a programmer agent and an Arch-Learner agent. The programmer agent's job is to simply read in the program mentioned in the previous paragraph and ask the Arch-Learner agent to cluster the rooms based on some criteria. The Arch-Learner agent uses the idea of learning from examples as presented by Winston [26]. Before the Arch-Learner agent can successfully complete his task, the user must teach him a concept by using positive and negative examples. In this example, I used the *train agent* menu in PENA to teach the Arch-Learner agent that public rooms cannot be adjacent to private rooms. How does it work? Suppose the user created two rooms, a kitchen and a bedroom, and placed them adjacent to each other. On the *train agent* menu, the user can then specify to the Arch-Learner agent that this is a negative example (Figure 14). The Arch-Learner uses a data structure called Thread Memory [27] to store all the knowledge it has about rooms in the program. For example, the kitchen would be represented as *room-public-kitchen* and the bedroom as *room-private-bedroom*. When the user presents the Arch-Learner with an example, the agent analyses the example and makes a rule of it. Coming back to our example, the Arch-Learner first calculates the distance between the kitchen and bedroom and concludes that they are adjacent.



◄ Figure 14. Training the Arch-Learner agent.

Next, the agent calculates the difference between the room's threads, concluding that one is public and the other is private. Finally, the agent receives input from the user that this is a negative example and adds a rule to his knowledge base which says private rooms should not be adjacent to public rooms. Once the Arch-Learner agent has grasped this concept of adjacency, he is able to take the list of rooms from the programmer agent and separate them into two clusters – private and public (Figure 15).

► Figure 15. Arch-Learner clusters private and public rooms.



7. Conclusions and future work

In this paper I surveyed the field of architectural programming, gave examples of existing computational approaches to the floor plan layout design problem – a key aspect of architectural programming – and suggested a model that takes a different approach to the problem – PENA, Programmatic Planning Environment for Negotiating Agents. PENA is in its early stages of development such that it is too early to extract any conclusive results. I believe PENA presents an interesting alternative to the widely popular optimization methodology, an alternative that might prove more useful to programming experts in practice. Some of the future work on PENA includes further development of agents' knowledge base and ways in which they use it to reason and make decisions. Furthermore, I would like to develop and implement more kinds of agents and to design the way in which they interact –cooperatively or competitively – to better resemble the interaction amongst experts in a real-world building project. Finally, I intend to extend the case study presented in this paper and to test PENA's capabilities on a real-world project.

Acknowledgments

I would like to thank Sarah Holton from Payette for all her help and support in better understanding the field of architectural programming and the job of the programming expert.

References

1. Peña, W. and Parshall, S.A., *Problem Seeking: An Architectural Programming Primer*, Wiley, New York, 2001.
2. Mitchell, W.J., Techniques of Automated Design in Architecture: A Survey and Evaluation, *Computers & Urban Society*, 1(1), 49-76.

3. Mitchell, W. J., An approach to automated generation of minimum cost dwelling unit planes, *International Technical Cooperation Centre Review*, 1975, 4(3), 116-139.

4. Homayouni, H., A Survey of Computational Approaches to Space Layout Planning (Unpublished).

5. Robert, S.F., A survey of space allocation algorithms in use in architectural design in the past twenty years, in: Hassler, E.B., ed., *Proceedings of the 17th conference on Design automation*, ACM, Minneapolis, Minnesota, United States, 1980, 165-174.

6. Yoon, K.B., A Constraint Model of Space Planning, *Topics in Engineering Series*, 1992, Vol. 9, Computational Mechanics, Southampton, UK.

7. http://www.payette.com/ [28-05-2009].

8. Mitchell, W.J., Steadman, J.P. and Liggett, R.S., Synthesis and Optimization of Small Rectangular Floor Plans, *Environment and Planning B*, 1976, 3, 37-70.

9. Sharpe, R., Marksjo, B.S., Mitchell, J.R. and Crawford, J.R., An Interactive Model for the Layout of Buildings, *Applied Mathematical Modeling*, 1985, 9, 207-14.

10. Flemming, U., Baykan, C.A., Coyne, R.F. and Fox, M.S, Hierarchical generate-and-test vs. constraint-directed search. A comparison in the context of layout synthesis, in: Gero, J.S., ed., *Artificial Intelligence in Design '92*, Kluwer Academic Publisher, Boston, 1992, 817-838.

11. Schwarz, A., Berry, D.M., and Shaviv, E., Representing and Solving the Automated Building Design Problem, *Computer-Aided Design*, 1994, 26(9), 689-98.

12. Jo, J.H. and Gero, J.S., Space Layout Planning Using an Evolutionary Approach, *Artificial Intelligence in Engineering*, 1998, 12, 149-62.

13. Medjdoub, B. and Yannou, B., Separating Topology and Geometry in Space Planning, *Computer-Aided Design*, 2000, 32, 39-61.

14. Michalek, J.J., *Interactive Layout Design Optimization: An Interactive Optimization Tool for Architectural Floorplan Layout Design*, M.S. Thesis, University of Michigan, Ann Arbor, MI, USA, 2001.

15. Nath G. and Gero, J.S., Learning while designing, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 2004, 18(4), 315-341.

16. Arvin, S.A. and House, D.H., Modeling Architectural Design Objectives in Physically Based Space Planning, in: Ataman, O., Bermudez, J., ed., *Media and Design Process: Proceedings of ACADIA 99*, Elsevier, Salt Lake City, Utah, 1999, 212-225.

17. Keatruangkamala, K. and Sinapiromsaran, K., Optimizing Architectural Layout Design via Mixed Integer Programming, in: Martens, B., Brown, A, ed., *Computer Aided Architectural Design Futures 2005*, Springer, Netherlands, 2005, 175-184.

18. Bier, H., de Jong, A., van der Hoorn, G., Brouwers, N., Heule, M. and van Maaren, H.: 2008, Prototypes for Automated Architectural 3D-Layout, in: Wyeld, T.G., Kenderdine, S., Docherty, M., ed., *Virtual Systems and Multimedia*, Springer Berlin / Heidelberg, 2008, 203-214.

19. Combes, L., Packing rectangles into rectangular arrangements, *Environment and Planning B*, 1976, 3, 3-32.

20. Fortin, G., BUBBLE: Relationship diagrams using iterative vector approximation. *Proceedings of the 15th conference on Design automation,* Las Vegas, Nevada, United States, IEEE Press, 1978.

21. Werkman, K.J., *Multiagent Cooperative Problem Solving Through Negotiation and Perspective Sharing*, PhD Thesis, Lehigh University, 1990.

22. Werkman, K.J., Multiple Agent Cooperative Design Evaluation Using Negotiation, in: Gero, J.S., Sudweeks, F., ed., *Artificial Intelligence in Design '92*, Kluwer Academic Publishers, London, 1992.

23. Sriram, D., Logcher, R., Fukuda, S., Computer-Aided Cooperative Product Development, *MIT-JSME Workshop*, MIT, Cambridge, USA, November 20/21, 1989.

24. http://en.wikipedia.org/wiki/Building_Information_Modeling [28-05-2009].
25. http://en.wikipedia.org/wiki/KQML [28-05-2009].
26. Winston, P. H., *Learning Structural Descriptions from Examples*, PhD Thesis, MIT, Cambridge, 1970.
27. Vaina, L. M., Greenblatt, R. D., The use of thread memory in amnesic aphasia and concept learning, *AI Working Paper 195*, 1979, Artificial Intelligence Laboratory, MIT, Cambridge.
28. http://en.wikipedia.org/wiki/Multi-agent_system [28-05-2009].

Yaniv (Junno) Ophir
Massachusetts Institute of Technology
Department of Architecture
77 Massachusetts Avenue, Bldg 3-409
Cambridge, MA 02139

junno@mit.edu