Abstract

CASH, STEVEN PATRICK. Bowyer: A Planning Tool for Bridging the gap between Declarative and Procedural Domains. (Under the direction of R. Michael Young.)

It has been show that software systems can benefit greatly by incorporating Artificial Intelligence (AI) techniques and practices into their functionality. Traditionally, there have been two large obstacles faced in attempting to apply AI techniques to commercial systems. The first obstacle is the gap between the largely declarative representations used by AI techniques and the largely procedural approaches used in commercial systems. The second obstacle is the gap between the skill sets and knowledge bases of the two domain experts with Artificial Intelligence researchers often lacking experience using specific APIs and run-time environments and commercial software developers often lacking significant Artificial Intelligence knowledge.

In this thesis I present Bowyer, a tool designed to address these two obstacles in the context of the integration of AI planning algorithms into 3D game engines. Bowyer bridges the gap between the declarative representations in a planning domain and the procedural framework of a virtual environment via the use of code generation techniques. Bowyer allows a user to specify a planning domain and then automatically generates a procedural representation of that planning domain for use in the virtual environment. Bowyer breaks the process of integrating a planning system with a game engine into three general steps. The first step involves the specification of a declarative planning domain in a representation based on the STRIPS planning language. The second step involves the generation of procedural representations of the planning domain operators and objects for use in the virtual environment through the mapping of virtual environment source

code to the planning operators and objects and using code generation techniques to create the full procedural representations of the operators and objects. The final step consists of integrating plans into the virtual environment by a) creating specific planning problems, b) automatically retrieving solutions to the planning problem from a planning service and c) executing the resulting plans in the virtual environment using the code generated by Bowyer. Bowyer's functionality allows planning researchers to integrate their planning research into virtual environments without the need to have extensive knowledge of virtual environment development.

Bowyer: A Planning Tool for Bridging
the gap between Declarative and Procedural Domains

By

Steven Patrick Cash

A thesis submitted to the Graduate Faculty of
North Carolina State University
In partial fulfillment of the
Requirements for the degree of
Master of Science

Computer Science

Raleigh, NC

2007

Approved By:

Dr. James C. Lester	Dr. Robert St. Amant

Dr. R. Michael Young
Chair of Advisory Committee

Biography

Steven Patrick Cash was born in Fayetteville, North Carolina on September 5th 1981 to George and Sandra Ansley. Patrick's mother, younger brother and his own last name changed from Ansley to his mother's maiden name Cash in 1994. Patrick lived in Fayetteville for the first 18 years of his life and graduated from Cape Fear High School in 2000.

After graduating from high school Patrick attended the University of North Carolina at Wilmington where he enjoyed living at the beach and the college life style. He graduated in 2004 with a Bachelors of Science degree in Computer Science and a Bachelors of Art degree in Physics.

After completing undergraduate work Patrick started his graduate study in computer science at North Carolina State University in the fall of 2004. He worked as a teaching assistant during his first two years of graduate school for courses in operating systems and introduction to Java. While completing his Master's course work Patrick selected artificial intelligence as his area of concentration for graduate research and joined the Liquid Narrative research group where he currently works under R. Michael Young.

After completion of his Master's degree Patrick plans to begin working on his Ph.D. in computer science at North Carolina State University while continuing to work at SAS Institute as a software developer.

Acknowledgements

I would first like to thank my committee: Dr. James Lester and Dr. Robert St. Amant for serving as my thesis committee members; and special thanks go to my thesis advisor Dr. R. Michael Young. I would like to thank the members of my committee for aiding me in my first attempt at major research work. Their knowledge and guidance was essential to the completion of this thesis.

I would not have been able to complete this thesis without the help of many others. I would like to thank the members of the Liquid Narrative group for aiding me in my research. Specifically I would like to thank: Arnav Jhala, Jim Thomas, Tommy Vernieri and James Niehaus.

I would like to thank Arnav Jhala for his countless conversations in the lab discussing ideas ranging from planning research and game design to the mundane topics of everyday life. He has provided me with insight into his own experience that has helped me answer many of my own questions. I would like to thank Jim Thomas for paving the way for Bowyer with the development of Bowman. Along with this I would like to thank him for his continued help with brainstorming of new ideas for planning tools as well as evaluation of our existing tools. I would like to thank Tommy Vernieri for his work on the Zócalo framework as well as helping me with numerous technical issues. Finally I would like to thank James Niehaus for his work on Crossbow and Longbow as well as always being a source of planning knowledge and an endless supply of good ideas.

I would like to thank: Yun Gyung Cheong, Byung Chull Bae, Jim Thomas, James Niehaus and Curtis Rawls for taking part in the Bowyer user study. Their input and feedback

was fundamental in the evaluation of Bowyer and its success at achieving the goals for which it was designed.

Finally, I would like to thank my family and friends for continuing to be supportive during my time in graduate school, helping my stay focused while also making sure I made time for life outside of it.

Table of Contents

List of Figures

List of Tables

1. Introduction

Software systems can benefit greatly by incorporating artificial intelligences techniques and practices into their functionality. These benefits generally allow the systems to be more flexible and robust, which is required as software systems continue to evolve and are expected to be more reliable and serve more important roles in society. Traditionally, there have been two large obstacles faced in attempting to apply artificial intelligence techniques to commercial systems outside of the research domain. The first obstacle has been the gap between the largely declarative based domain of artificial intelligence techniques and the largely procedural based domain of most commercial systems. The second obstacle, that may be largely a reflection of the first, is the lack of knowledge concerning industry established design paradigms and software engineering techniques by artificial intelligence researchers and conversely the lack of artificial intelligence knowledge by most software developers.

A more specific example of these obstacles can be seen in the difficulty faced when trying to integrate artificial intelligence planning techniques, which generally utilize declarative knowledge representations, into applications like virtual environments such as commercial video games, which generally are based on procedural representations and control flow. Adding to this difficulty is the usually large learning curve encountered when working with systems of the size needed to create realistic virtual environments. Typically planning researchers have a high level of knowledge concerning the techniques and practices used in planning research but little to no knowledge of the techniques and practices used in virtual environment development. Conversely, game designers typically have a high level of

knowledge with respect to constructing virtual environments but little to no knowledge about planning techniques.

Virtual environments such as video games are beginning to increase their need for improved artificial intelligence that has higher integration into the environments as the ability of graphics to create immersive environments begins to level off while the users' expectations continues to climb [15] [16] [18]. This thesis presents Bowyer, a tool aimed at overcoming these two obstacles by bridging the gap between the declarative based planning domains and the procedural based virtual environment domains as well as aiding planning researchers unfamiliar with virtual environment development in integrating their planning research into virtual environments.

Bowyer's approach to solving these two problems can be broken down into distinct steps used to translate planning domain representations into virtual environment representations. First Bowyer allows the user to visually build a planning domain representation by specifying the types, literals, objects and operators that make up the planning domain. Next, in a similar manner Bowyer allows the user to specify a client representation of the virtual environment that will be used in the translation process. This client is a representation of a lightweight module responsible for communications involving the execution of the plan and contains the generic planning domain representations that will be used by Bowyer. Along with this client specification, a code library of source code modules to use in domain translation is created using Bowyer's user interface. After both domains have been specified the next step is for the user to map planning operator and object attributes to their virtual environment representations and once this is done generate the

virtual environment representations of the planning operators and objects. Once these representations have been generated the final step is to test the generated representations using Bowyer's built-in planner support and integration into the virtual environments using an existing planning framework [29][33]. Bowyer also provides some additional functionality to aid the user in utilizing this framework to interact with the virtual environments.

In this thesis I outline the existing research drawn upon by Bowyer, present an outline of Bowyer's functionality and an example of Bowyer in use. I also present the findings of a pilot study used as an initial evaluation of Bowyer's ability to achieve the goals it was designed for and a discussion of Bowyer in its current state and plans for future work on the tool. In section 2 I present a brief overview of relevant background and related work. Section 3 contains an overview of Bowyer's functionality, describing Bowyer's approach to surmounting the two obstacles described above. In section 4 I walk through a sample usage of Bowyer, providing an example for each step in the process of translating a planning domain into its virtual environment domain representation. In section 5 I present the findings of a pilot study used for an initial evaluation of Bowyer from a target user prospective. Finally, in section 6 I analyze Bowyer's ability to bridge the gap between declarative and procedural domains and aiding planning researchers as a whole, discuss future work and include some concluding remarks.

2. Background

Bowyer draws upon work from many areas spanning from artificial intelligence research to commercial game development. In this section I present background information from planning domain representations, Unreal Tournament®, the virtual environment used for the example provided in section 4, code generation, tools to aid in planning research and a tool used to generate game code from user specified scenarios.

2.1 Artificial Intelligence Planning

Planning is a well established field in the area of artificial intelligence with over three decades of ongoing research [11][27][31][32]. Bowyer's planning support is based on techniques that have been proven over this time period. The aspects of planning that Bowyer addresses are primarily planning domain specification and planning problem specification. Bowyer also provides functionality to interact with planners allowing users to specify the planning domain to use, the planning problem to use and some of the aspects the planners use to search for a plan.

Bowyer's planning domain and planning problem specification methods are based on the methods used by the Zócalo planning framework [29] and its predecessor [33]. The methods used by Bowyer were developed to work with Crossbow, the default planner in the Zócalo framework. Crossbow [29] is the C# implementation of the Longbow planner. Longbow [35] is an extension of the UCPOP planner that adds hierarchical planning methods to UCPOP. As a result Bowyer's planning domain and problem specification are based on the STRIPS planning language. STRIPS [6] is a system that was developed early in the field

of planning research and provided the groundwork for large amounts of planning research based on its planning knowledge representation.

STRIPS represents the planning domain as a set of operators that represent the possible action is the domain. Operators is STRIPS are a triple {P, A, D) which represents a set of preconditions (P) and an add (A) and delete (D) list. Preconditions represent the conditions that must hold true in the world for the execution of the operator to be valid. The add and delete lists represent the conditions or facts in the world that are added to the planning domain or deleted from the planning domain based on the operator's execution.

STRIPS represents the planning problem as a set of two states that are in turn represented by a set of literals. The first state is the initial world state which represents the world at its current or initial state before the plan takes place. STRIPS uses the closed world model which requires every literal that is true in the world to be specified in the initial state and all other literals are considered by default false. The second state in a planning problem represents the goal state for the plan. This state represents the desired state of the world once the plan has been executed. The generated plan is responsible for transforming the world from the initial state to the goal state through a set of actions represented by the operators in the domain.

Bowyer's planning domain and planning problem specification are very similar to the specification used by STRIPS, UCPOP and Longbow. The planning domain used by Bowyer is defined by a set of operators that represent the possible actions in the planning world. Similar to STRIPS, Bowyer's operators are specified using a triple (P, C, E) which represent a set of preconditions, constraints and effects. Preconditions, like in STRIPS,

represent the conditions that must hold for the execution of the operator to be valid. Constraints represent the constraints on the values assigned to the operator's parameters. Effects represent the change to the world that occurs based on the execution of the operator and are roughly equivalent to the add and delete list used by the STRIPS representation.

2.2 Unreal Tournament 2004®

Unreal Tournament 2004® (UT2004) is a popular commercial video game in the first-person-shooter genre created by Epic Games®. It provides a high-level immersive virtual environment with high-end graphics, sound, artificial intelligence and simulated physics built into the game. While many other current commercial games may have these features, UT2004 was selected as the first virtual environment to integrate Bowyer into based on its modification or "mod" support functionality. Other current commercial games also provide similar functionality and can be used with Bowyer in a similar way to UT2004.

UT2004 modification functionality allows a user with development knowledge to modify the game, or create new games, using the functionality the game designers provide through the UnrealEd® level designer and Unreal Script® game scripting language. This functionality allows the game client, which is required by Bowyer and Zócalo, to interact with the virtual environment created by the game engine.

2.2.1 Unreal Game Engine

The unreal game engine is the software system behind the UT2004 game and is responsible for creating the virtual environment in which the game is played. It controls the low level functionality including: rendering of the graphics to the screen, sounds, simulated physics and networking support for the virtual environment [17][24][25]. While the unreal

engine supports the low level functionality that is required to create the immersive virtual environment in the game, the high level functionality of the game including game rules, non-player character (NPC) artificial intelligence, etc. are controlled by Unreal Script code that runs in a layer on top of the Unreal game engine.

2.2.2 Unreal Script

Unreal Script is a scripting language created by Epic Games® that, as discussed above runs on top of the Unreal game engine and is used to control game rules, game characters, player interaction, high level networking and several other facets of the gaming experience. Unreal Script is a fairly robust programming language that provides many of the same features as mainstream programming languages [26]. Unreal Script allows the user to override or extend existing functionality in the game world; it is this functionality [2] that is taken advantage of by the Zócalo game client used by Bowyer.

2.3 Code Generation

Code generation is an establish field of research in computer science and has proven to be a useful technique in research based and commercially viable systems. Code generation covers a large area of applicability including: code generation for embedded devices, code generation done by compilers to generate machine code, and generation of source code ranging from macro expansion to generation of entire applications and complex systems. This section will only address the form of code generation for human readable programming languages given that this is the type of code generation Bowyer utilizes.

The general approach taken to generate code is largely dependent on the scale of the code that is to be generated. The most common approaches can be broken into roughly five

groups: code monger, inline/mixed generation, partial class generation, layer generation and full domain language [12]. A generator using the code monger approach takes source code as input and modifies parts of the source code based on pattern recognition. A generator using the inline/mixed generation approach also take source code as input and if special symbols are found in the code the symbols are replaced with a corresponding source code module. A generator using the partial class generation approach uses an establish pattern for a source code class and adds or modifies code in the class where the pattern differs from class to class. Generation using the layer approach also uses patterns in code but at a higher level of abstraction, representing an entire application level instead of a single source code class. Finally, a generator using a full domain language is designed to generate an entire application for a specific domain and uses the assumptions available within the domain to raise the level of abstraction in the language so that the software can be specified at a higher level.

Bowyer's code generation is based on a partial class generation approach to code generation. Bowyer uses a code library of small code modules or code "chunks" that can be specified by the user using Bowyer's interface. These code modules are mapped to planning operators and objects and an XML code representation is created for the planning object or operator. XML has been shown to be a viable format for representing source code structure [36]. XSLT templates [13] are then used to transform the XML code representation into the desired output source code. Bowyer's specific process for code generation is explained in the Bowyer overview section. The rest of the section will focus on code generation research related to this approach to code generation.

Captool [19] is a tool designed to bridge the gap between software engineers and physicist in developing intelligent instruments. Normally software engineers and physicist do not speak the same "language" so the cooperative development of tools using both their skill sets can be difficult. The aim of this tool is to allow the software engineers to develop small reusable code modules and then allow the physicist to create intelligent tools by creating components using these modules. The code generation technique used by this tool is very similar to the generation technique used by Bowyer with the instrument being modeled by the domain expert, the physicist, and the Captool tool creating an XML representation of the code for the instrument which uses XSLT templates to transform the XML into source code. This tool also shares an additional goal with Bowyer of choosing to use this specific code generation technique so that generation of additional languages only requires the addition of new XSLT templates and new code modules for that language.

Finally, the MOmo compiler [1] is another tool that uses the XML representation and XSLT template transforms approach to code generation. The compiler is based on the idea of using modeling languages at a higher level of abstraction then high-level programming languages. The models are given to the compiler as XMI representations of the modeling language. The compiler then parses the XMI input and builds internal XML representations of the models, which includes breaking the models down into smaller pieces. Once these internal XML representations have been built the compiler then uses XSLT style sheet transforms to generation the implementation source code for the systems modeled in the input XMI modeling language.

2.4 Planning Tools

The planning tools covered in this section are, like Bowyer, tools that were designed to help planning researchers conduct their planning research. They are developed to help planning researchers create and test their planners by allowing the planning researchers to more efficiently: communicate with their planner, create planning domain and planning problem specifications and help automate and formalize other common planning research tasks. I make this distinction to differentiate these planning tools from tools that incorporate planning techniques as an artificial intelligence tool to accomplish other tasks but are not aimed at improving planning research techniques. Most tools of this nature center around improving the ability to specify a planning domain and planning problem, validate a given specification or extract information from a planning domain specification.

GIPO [22] is a system designed to aid planning researchers in defining planning domain models. GIPO's functionality includes a graphical interface to define planning domain models, tools for checking the validity of the developed domain models, tools to validate the domain models against existing plans, import and export of planning domains using the PDDL [5][7][9] planning language as a storage format and an interface for integration with external planning algorithms.

The process of domain specification is completed over a series of steps. The first step is to identify the types of objects that characterize the domain. For each type of object the valid states each object can be in must be specified. It is assumed that the application of operators to the objects is what moves the objects between these states and each possible change must be specified in the object. Operators are then specified by selecting the subset

of possible transitions for that operator from the list of available transitions based on the operator's type. Once a planning domain has been defined and verified an external planning algorithm is used to find a plan. GIPO also allows the user to test plans using either a single step based interface or an interface that allows animated execution of the plan in a graph animation. GIPO II [23] is an extension of the work done on GIPO that allows the specification of hierarchical domain models and includes more domain validation support including validation support for the added hierarchical specification.

VITAPlan [30] is a visual tool used for the highly adjustable planning (HAP) system. The adjustable nature of the system allows the user to adjust and fine tune the parameters used in searching for plans, such as heuristic used and value weightings. ViTAPlan allows the user to configure the parameters but also implements a rule system, which analyses the problem and applies a set of rules, developed from machine learning techniques, to automatically configure the parameters for each problem and provides advice to the user on possible configurations. This advice takes the form of an analysis of the problem and listing the triggered rules in the rule system.

Similar to GIPO, ViTAPlan also supports an animation of the execution of a plan, checking the consistency of a plan, uses a graphical interface for specifying planning domains and planning problems and generates the PDDL specifications of the planning domain and planning problem. In contrast to GIPO the planning domain is built by defining the objects, predicates and the relations between them. The operators in the domain are represented as frames with three columns. The center column represents the objects in the operator that correspond to the operator's parameters. The left column is made up of

predicates and represents the operator's preconditions, while the right side column is made up of predicates and represents the operator's effects on the state of the world after execution of the operator. This representation roughly corresponds to the STRIPS representation discussed previously.

Bowman [28] is a system that was developed to aid planning researchers, game designers and other users interested in utilizing planning for interactive narrative. The overall goal of Bowman is to involve the user in a collaborative authoring environment with a planner allowing the author to adjust how the planner creates the plan to allow the plan to be tailored to the user's interest. This is achieved through iterative refinement of the returned plan allowing the user to modify the parameters used to develop the plan.

The currently implemented version of the tool allows the user to specify a planning domain using types, literals, objects and operators, similar to Bowyer planning domain specification. Along with providing a graphical interface for specifying the planning domain Bowman provides functionality to interact with the planning modules of the Zócalo framework. Currently the only implemented metric in place to allow the user to modify the plan returned by the planner is to select the heuristic used to search for a plan. The additional functionality planned for future work for Bowman is discussed in the next paragraph.

In addition to the currently implemented functionality Bowman is planning to include additional support to allow the user to customize the manner in which the planner searches for a plan. This functionality centers on what the author calls the Domain Elaboration Framework (DEF). DEF is a "domain meta-theory that allows the authors to add detail to classical planning domains to enable expressive problem definition and reasoning about

plans."[28]. DEF will be able to accomplish this by using a grammar that consists of types, dimensions, weights, and measurements to add additional tuning parameters to the planning objects, currently only the type functionality is implemented. The DEF formalization will be used to specify the user's preferences in returned plans to allow the user to tailor the plans to the situation and narrative characteristics that are desired.

Bowman and Bowyer were designed to be unified into one tool in later versions and as a result shared similar user interfaces for the areas in which their functionality overlaps. These two areas are planning domain specification and planner interaction through the Zócalo planning framework. These two tools are designed to be integrated into a complete tool to aid users in working with the Zócalo planning framework to aid in integrating planning techniques and interactive narrative into virtual environments.

Along with the tools covered above, there are tools in other areas of planning research that aid planning researchers in their research. The majority of these tools provide different forms of verification and validation support for planning research. One example is the VAL tool [14] that is used to verify plans specified in the PDDL planning language. VAL provides a report of the plan validation process with information detailing why a plan failed as well as advice on how the plan can be repaired. Another example is the TIM tool [8] that is used to extract generic types, state invariants and other attributes from planning domain specifications. This aids planning researchers in debugging their planning domain specifications and increasing the efficiency of their planners.

2.5 ScriptEase

ScriptEase [20] is designed to be a visual tool for generating scripts for a commercial computer role-playing-game. More specifically ScriptEase is a tool that was developed for the Neverwinter Nights® (NWN) role playing game and generates NWScript® the Neverwinter Nights proprietary scripting language, similar to UnrealScript® for UT2004®, to create scripted sequences to control non-player characters in the game world.

ScriptEase consists of two basic components, the Atom Builder and the Pattern Builder. The Atom Builder is designed to be used by programmers to create small *atoms* of scripting code that represent common actions in the scripted sequences in NWN. Atoms can be viewed as a data type consisting of a name, type, parameters, description information, and the source code for the atom. The Pattern Builder is designed to be used by designers (non-programmers) and uses the atoms built by the Atom Builder to create patterns of actions that are common in NWN scripted sequences referred to as situations. The process for generating these scripted sequences follows a series of steps. First a library of atoms to be used for code generation must be defined. After this library has been created the user creates situation patterns with the atoms. Once this is complete the code for the situation can be generated and the NWN toolkit Aurora® can be used to compile the code into game executable code. Scripts in NWN are associated with the objects in the game world and each object has slots to add scripts to that object. The game play in NWN is event driven and these scripts are called when the object's actions trigger some event causing the scripts to execute.

ScriptEase's approach to generation of game code using code modules and common patterns is similar in concept to the approach Bowyer takes to generate code. The use of

patterns to generate common source code is prevalent in code generation techniques in general. Bowyer's differentiation from ScriptEase can be seen in its planning support, through its visual specification of planning domains and mapping of planning objects and operators to virtual environment representations for execution of plans in the virtual environment. Bowyer also utilizes an approach to code generation, discussed in the next section, which allows new languages and new virtual environments to be supported with the addition of XSLT templates, a code library and client specification without the need to rebuild the Bowyer application code base to support the new virtual environments. ScriptEase requires significant changes to its code base to be able to support generation of scripts for new virtual environments including changes to its file format for storing atoms and patterns, it game specific code generation functions and other game specific code in its code base.

Bowyer bridges the two distinct domains of planning research and virtual environment development and accordingly Bowyer draws from work in a wide variety of fields.

3. Bowyer Overview

This section provides an overview of Bowyer's functionality, describing the manner in which Bowyer's implementation allows the user to integrate planning into virtual environments. Bowyer allows the user to: create a specification of a declarative planning domain, create a specification of the procedural virtual environment domain, create a mapping between the planning domain operators and objects and the virtual environment

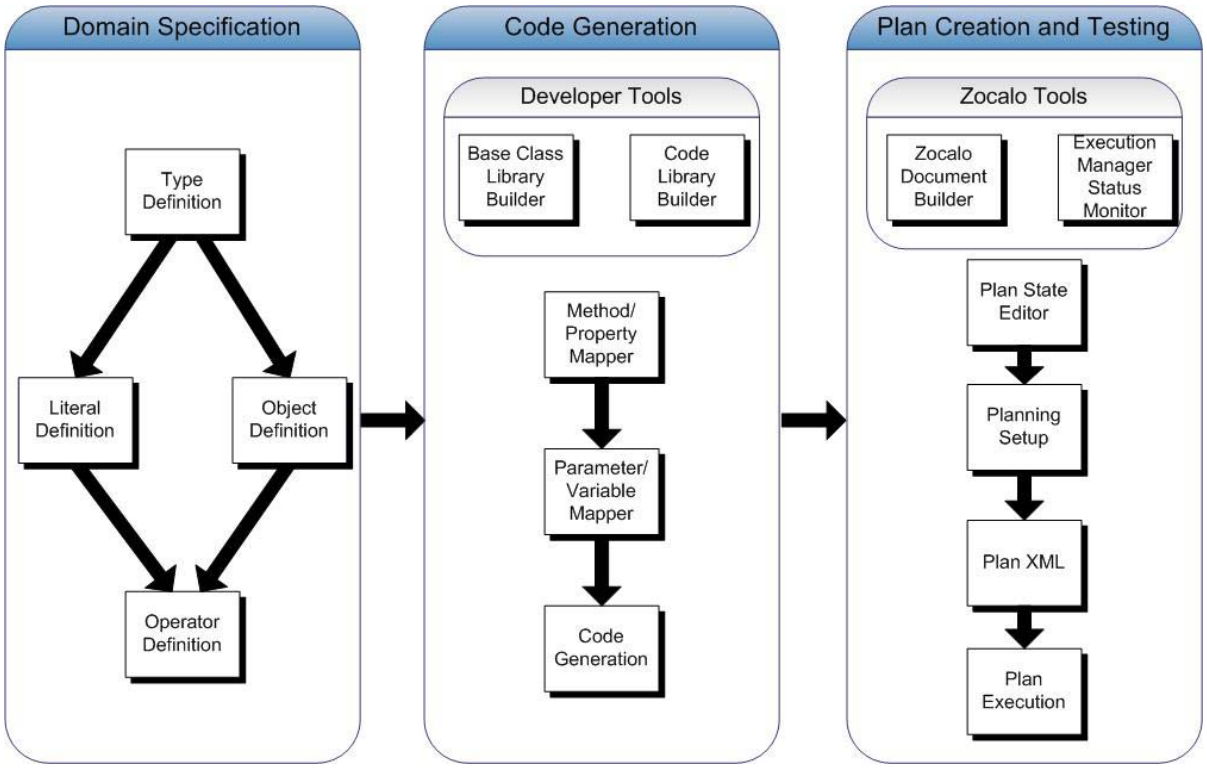

Figure 3.1 Bowyer Overview

code library and generic classes, generate the virtual environment source code for the procedural representation of the planning operators and objects, build the XML documents used by the Zócalo planning framework, specify a planning problem, retrieve a plan from a

planner and test the generated procedural representations in the virtual environment by using the plan returned by the planner, Bowyer and the Zócalo planning framework. Figure 3.1 represents the logical grouping for the three sections of Bowyer's functionality.

3.1 Planning Domain Specification

Bowyer allows the user to create a planning domain specification using its graphical user interface to create a STRIPS like domain specification. This sub-section presents an overview of the process used to specify the planning domain by creating the types, literals, objects and operators that exists within the planning domain.

3.1.1 Type Definition

Bowyer allows the user to specify the types of objects in the planning domain. All objects in the planning domain have to be of a type that is defined using the graphical user interface provided by Bowyer. Types in Bowyer are defined in a hierarchical manner with the root of the hierarchy being the type "anyThing" and all types are directly or indirectly derived from this type. Types in Bowyer are defined using the *Type Editor* interface and simply consist of the type name.

3.1.2 Object Definition

Objects in Bowyer represent the physical objects and locations in the planning world that will be used in the planning process. Objects in Bowyer consist of a name, a description and can have one or more types assigned to it. Bowyer allows the user to add, edit and delete objects from the planning domain using the *Object Editor* and *Object Editor Dialog* interfaces.

3.1.3 Literal Definition

Literals in Bowyer represent the literals in the planning domain. The literals defined in this interface will be used in the definition of operators in the next stage of Bowyer's interface as well as the planning problem used to send to a planner. Literals are used as the operator's preconditions, constraints and effects and to define the initial and goal states in the planning problem. Literals in Bowyer consist of a name, description and a set of arguments for the literal represented by their types. Bowyer allows the user to add, edit and delete literals from the planning domain using the *Literal Editor* and *Literal Edit Dialog* interfaces.

3.1.4 Operator Definition

Operators in Bowyer represent the actions that are possible in the planning domain. Operator representations consist of: a name, preconditions, constraints, effects and a set of parameters similar to the way operators are represented in the commonly used STRIPS planning language. Preconditions represent the conditions that must hold true in the planning world's current state for the operator to execute. Effects, also referred to as postconditions, represent the consequences or effects in the world caused by the operator when it has completed execution. Constraints represent the conditions that must hold for the operator to be valid; often these are constraints on the valid types of the values assigned to the parameters that are used by the operator. Preconditions, constraints and effects are represented as lists of literals. Operator parameters represent the variable properties of the operator that will be bound to world objects when the operators are executed.

Bowyer allows users to add, delete and modify operators in the planning domain using the *Operator Editor* interface to create and delete operators as well as add and delete

operator parameters. Bowyer allows the user to add and remove literals for the operator's constraints, preconditions and effects using the *Operator Edit Component* interface. The operator's constraints, preconditions and effects are edited separately using the *Operator Edit Component* interface which allows the user to add any of the literals defined in the domain to the set of literals that make up the constraints, preconditions or effects as well as bind types or objects to each of the literal's arguments.

3.2 Virtual Environment Domain Specification

In this sub-section I present an overview of how to specify the client information for the virtual environment client representation used by Bowyer. This includes specifying a client type and the generic operators and objects representations provided as action classes in the Zócalo game client that will be used as parent classes for the operator and object classes generated by Bowyer. Also the process for building a client code library for a specific virtual environment is discussed.

3.2.1 Client Specification

Clients in Bowyer represent the virtual environments that can be used by Bowyer and have a Zócalo game client implementation associated with them that is required by Bowyer to execute plan steps in that virtual environment. More specifically the Bowyer client represents that actual Zócalo game client implementation for that virtual environment. Bowyer's *Client Configuration* Dialog allows the user to specify the client attributes of the Zócalo game client. Clients are specified by entering the client name, client XLST template files location; client base class description file location and the client source code file extension.

The client name is the name used to represent the virtual environment client in Bowyer; this name will be present in all of the client type menus. The client XSLT template files location is used to specify the location of the XSLT template files used for code generation for that client. The client base class specification file location refers to the location of the XML file that specifies the generic classes from which all classes generated by Bowyer will be derived. This document is created by the *Base Class Library Builder* interface discussed in the next section. Finally the client source code extension is the file extension used for source code files in the programming or scripting language used in the virtual environment. This is used by Bowyer to add the correct extension to the generated source code files.

3.2.2 Client Base Class Definition

The client base classes represent the generic operator and object classes in the Zócalo game client, referred to as action classes and world objects respectively. This specification of the client's base classes should only have to be created once for each virtual environment and is expected to be done by the developer of the Zócalo game client for that virtual environment. It is then reusable by anyone using the Zócalo game client for that virtual environment. This is one of the steps taken in Bowyer's functionality to help aid non-developer users of Bowyer with the code generation process. The definition of the client's base classes is necessary for the code generation step to allow the methods in the base classes to be mapped along with code library methods and properties, to the planning operators and objects. The process of specifying base classes consists of specifying the name of each base

class as well as the signature for all of the methods in each base class. A method's signature consists of the scope, return type, name and parameters for that method.

3.2.3 Code Library Building

Although some functionality for the generated source code classes is inherited from the virtual environment's Zócalo game client base classes, additional functionality will be needed in the generated operator and object representations. This additional functionality is obtained from code modules that are saved in source code libraries using Bowyer's interface. Each virtual environment must have its own code library built using the programming language that corresponds to that specific virtual environment. As with the base class definition the code library is designed to be sharable between users and specified by developers for the virtual environment such as the developer of the Zócalo game client for that environment. This portability of the code library is another feature used to aid non-developer users in the code generation process. The code library for each client is built, using the *Code Library Builder* interface, by adding method and property specifications that can be used to specify possible operator and object functionality. It is expected that the code library for each environment will grow and become more robust over time with the addition of new methods and properties.

Methods are specified by entering the scope, return type, name, parameters, code body and a description for the method. Properties are specified by entering the type, name and default value of the property. Often the property value will be changed after the code using that property is generated. Developers entering these properties can enter hints to the possible values the property should take instead of entering an actual value for the property.

3.3 Mapping Between Declarative and Procedural Domains

This sub-section gives an overview of the process of mapping from planning operators and objects attributes to virtual environment code. This includes mapping code modules to planning operators and objects and mapping code variables to operator parameters. After the mapping is complete the virtual environment code representation of the planning operators and objects can be generated, viewed, edited and saved for use in the virtual environment.

3.3.1 Method and Property Mapping

The main step in translating declarative planning operators and objects into their procedural virtual environment representations is to map code methods and properties to operators and object. The mapping process is completed using the *Method/Property Mapper* interface. For planning operators the mapping consist of mapping the literals for the preconditions, constraints and effects to their client code method representations as well as mapping the code representations for the operator's execution, the code that actually performs the operator's effects, to the operator as well as any code that should be in the operator's initialization code. For planning object translation into procedural representation, Bowyer allows the user to map any methods or properties to the planning object that reflect the object functionality. The user assigns mappings to the planning operators and objects until the mapping for each has been completely specified. After the mapping for all of the operators and objects has been completed the next step is the parameter to variable mapping stage for operators.

3.3.2 Parameter to Variable Mapping

After all of the methods have been added to an operator all of the variables used in each of the method's parameters must be mapped to the operator's parameters. This is necessary because the variables used in the code represent the world objects and location in the game world and the operator's parameters are used by the action classes to find the correct virtual environment world objects to map to these action class variables. A simple one to one mapping is created for each method's parameters using the *Parameter/Variable Mapper* interface

3.3.3 Code Generation

Code generation in Bowyer uses the mapped relationships defined in the previous two steps to create an XML specification of the planning operator or object to be generated. This XML specification is then sent to be processed by that specific client's XSLT templates to generate the source code to be used in that client's virtual environment. The generated source code is then displayed to the user in the *Code Generation* interface so that any

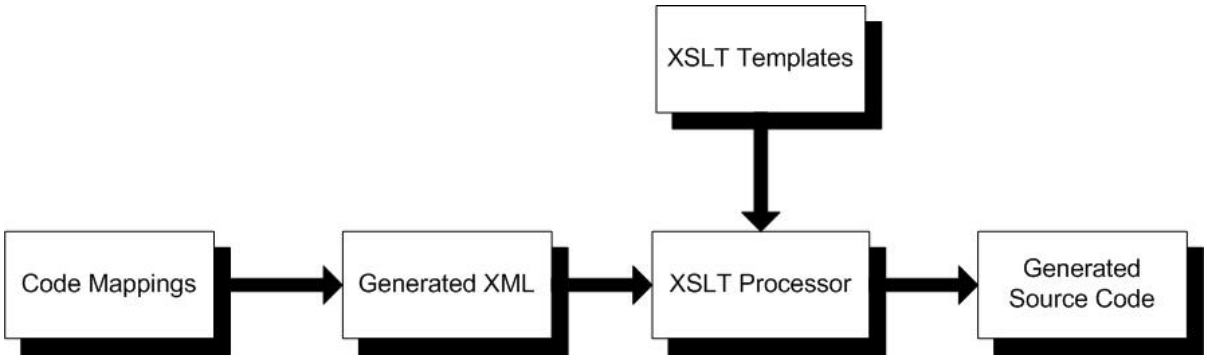

Figure 3.2 Code Generation Process Flow

changes can be made and the source code can be saved to file to be used in the virtual environment.

This code generation technique was selected to allow Bowyer to generate code for additional clients and additional programming languages without the need to modify the Bowyer code base. Additional generation languages and game clients can be supported by Bowyer through the following steps. First the client's base classes must be specified and a code library must be built for that client as discussed earlier. Next, the XSLT templates for the client must be created and put in the correct location under the Bowyer install directory with the top level control template for the client named as [*Client name specified in Bowyer's client type menu*]_ControlTemplate.xsl. Finally, the new client must be specified in the *Client Configuration* dialog box as discussed earlier in this section, giving the location of the base class definition and XLST templates. This allows Bowyer to know what set of templates to call to handle the code generation for a specific client.

3.4 Integration into Zócalo Planning Framework

In this sub-section I present an overview of the existing Zócalo planning framework [29] and the manner in which Bowyer integrates into this framework to test the generated procedural representation of the planning operators and objects. This includes a brief overview of the components that make up the Zócalo framework, functionality Bowyer provides for generating the XML documents needed by Zócalo and functionality Bowyer provides for planner interaction and plan execution utilizing the Zócalo framework.

3.4.1 Zócalo Planning Framework Overview

The Zócalo planning framework is a web service based framework made up of several independent components. It is designed to facilitate planning support in virtual environments by providing a mechanism for the virtual environment to request a plan from a planner and have the Zócalo framework control the execution of the plan within that environment. Figure 3.3 is a high level overview of the Zócalo framework; a more in-depth discussion including plan integration into the virtual environment experience and the Zócalo communication process flow can be found in [29].

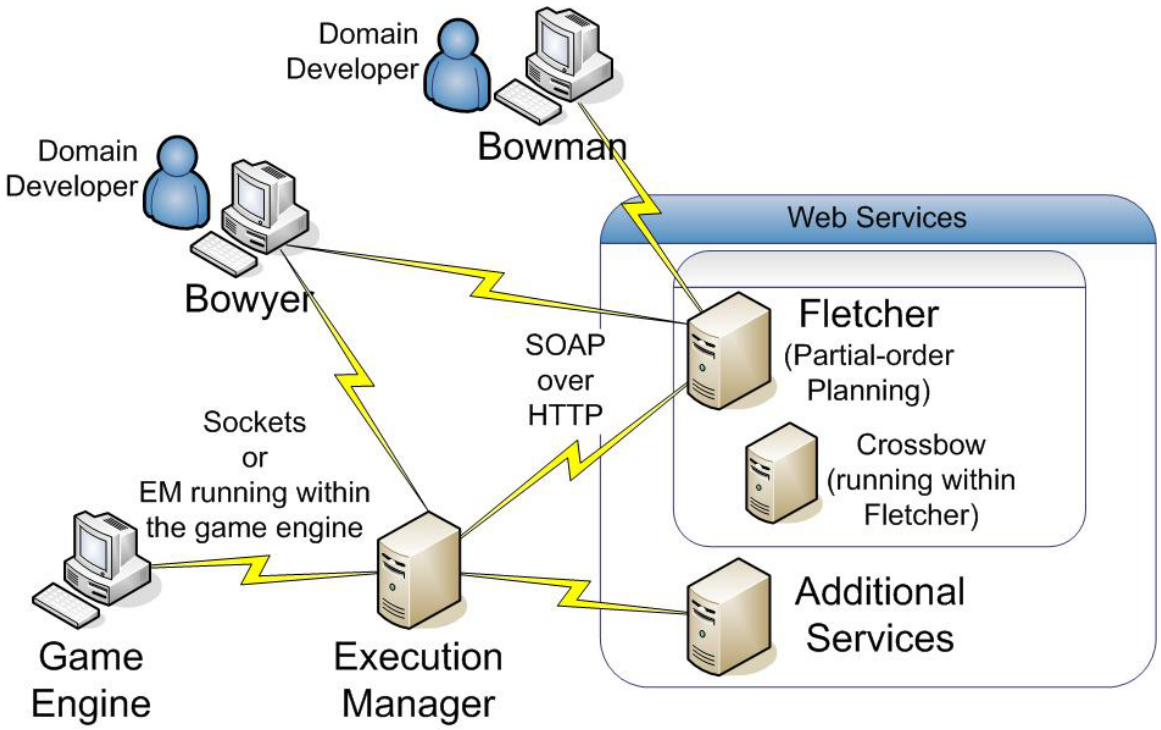

Figure 3.3 Zócalo Overview

3.4.1.1 Game Client

The Zócalo game client component is a light weight client for virtual environments that contains the functionality needed to communicate with the rest of Zócalo, more

specifically to communicate with the Execution Manager component. The Zócalo game client's main responsibilities are: connection to Zócalo, request of a plan from Zócalo, executing a step when a execute step message is received from Zócalo, communicating back to Zócalo the success or failure of a step and finally handling mediation support in the plan.

3.4.1.2 Execution Manager

The Execution Manager serves as the main hub for the Zócalo framework and controls plan execution. The Execution Manger's main responsibility is managing the execution of the plan returned from the planner. By default the planner used by Zócalo is the Crossbow planner discussed below. The Execution Manager maintains a directed acyclic graph (DAG) representation of the plan used for execution, it also maintains representations of the planning domain and plan mediation data structures and updates this representation as needed during the execution of a plan.

3.4.1.3 Crossbow

Crossbow [29] is a C# implementation of the Longbow partial-order planner developed by Young [34][35]. Crossbow is used as the default planner by the Zócalo framework and is used in combination with Fletcher, discussed below, to provide the planning support for the Zócalo framework. Although Crossbow is used as the default planner in Zócalo, any planner that can support the web-service interface that Zócalo requires can be used as part of the Zócalo framework.

3.4.1.4 Fletcher

Fletcher [29] is a web-service wrapper for the Crossbow planner used by Zócalo. Fletcher provides the web-service interface for the Crossbow planner that Zócalo requires

planners to support. In addition to serving as a wrapper to the Crossbow planner, Fletcher also provides additional functionality for planning support including working with the Zócalo plan mediation components discussed below.

3.4.1.5 Mediation Components

Crosswind [29] is a tool used by Zócalo to provide reactive mediation support for the plans used with the Zócalo framework. Kyodo [10] is a tool used to provide proactive mediation support for the plans used with the Zócalo framework. Bowyer does not currently utilize the mediation support in the Zócalo framework using these two planning tools but future integration is possible.

3.4.1.6 Planning Tools

Bowman [28] is a planning tool used with Zócalo for narrative planning support. Bowman, as covered in the background work section, is designed to involve the user in a collaborative authoring environment with Bowman allowing the author to adjust how the planner creates the plan.

The Bowyer tool, discussed in this thesis, is also classified as a planning tool to support the Zócalo planning framework.

3.4.2 Zócalo Document Specification

Zócalo uses several XML documents to specify information such as the planning domain, planning problem, environment actions, user actions, etc. that will be used in the planning process. Bowyer provides a way to easily create these documents using its *Document Builder* interface and corresponding *Document Builder Dialog* interfaces for each document type. These interfaces are used to guarantee the generated XML documents will

match the correct XML schema and correct information needed so the user will not have to create the documents by hand.

3.4.2.1 XML Document Creation

Bowyer's *Document Builder* interface allows the user to select a document type from the list of available document types used in the Zócalo framework. The user can then create a new document or edit an existing document. If the user chooses to create a new document the *Document Builder* dialog for that document is displayed with the fields from the Bowyer planning domain populated and the fields corresponding to the document are empty. The document can then be created and saved using the interface provided. If an existing document is opened then the Bowyer planning domain fields and the document fields are populated and the document can be edited in the same manner that the new document is created. When the document is saved the existing document will be overwritten with the new specification.

The Action Specifier document is used to specify the possible constants, constants groupings and actions groupings that can be initiated by the player in the virtual environment or by the virtual environment itself while the plan is executing.

The Domain document is used to specify the operators that are part of the planning domain used by the Zócalo framework.

The Environment State document is used to specify the initial state of the environment through a list of literals that hold true in the world. This is based on the closed world assumption used by the Zócalo framework.

The Problem document is used to specify the problem to be solved by the planner. This document consists of the literals that specify the initial state of the world and the literals that specify the desired goal state of the world.

The Set State and Manage Plan document and the Manage Plan document are the documents used at the creation of a Zócalo session to specify the planning domain, initial state of the world and desired goal state of the plan.

The Set State and Manage Plan document specifies the domain operators, the initial state literals and the goal state literals.

The Manage Plan document does not have to specify the initial state because it does not set the state of the world for the Zócalo framework. It is used to specify the planning domain operators and the goal state literals.

3.4.3 Planner Interaction

Bowyer provides functionality to allow the user to graphically setup a planning problem and planning domain, connect to a planner, request a plan, and display the returned plan found by the planner in a tree and graph view form.

3.4.3.1 Planner Setup

. Bowyer's planner support is based on the planner support that the Zócalo framework provides and therefore requires that planners to support the same web service interface that Zócalo requires. To allow Bowyer to use an existing planner the user must specify the location of the planner implementation. Bowyer allows the user to choose between one of four options for planner location. The first choice represents using Bowyer's internal planner which is an implementation of the Crossbow planner. The second choice

represents a planner running on the same machine on which Bowyer is running. The third choice represents using the planner that the Liquid Narrative group maintains at a specified location. The final choice allows the user to specify any other location, via a URL, that has a planning web service available.

The selection of planner location is assumed to be done at the beginning of a Bowyer session using the *Configuration* dialog which also allows the user to specify the Bowyer installation base directory. This is used by Bowyer to locate other configuration information used by Bowyer.

3.4.3.2 Domain and Problem Specification

Before using a planner to find a plan the planning domain and planning problem must be specified for use by the planner. Bowyer gives the user a couple of options for the specification of both the domain and planning problem. Bowyer allows the user to simply specify the location of a domain and problem document that will be sent to the planner or Bowyer allows the user to use the domain maintained by Bowyer as the planning domain and to specify a planning problem created with the Bowyer *Plan State Editor* interface. This interface allows users to specify the literals that make up the initial and goal states of the planning problem in the same interface the user can define the literals that make up an operator's preconditions, constraints and effects.

Bowyer is designed to primarily be used with plan space planners; accordingly Bowyer allows the user to specify existing incomplete plans to use as the start node for planning search instead of using a planning problem specification. Additionally, Bowyer allows the user to select the search heuristic that is used in the search for a plan, given that

the planner being used implements selectable heuristics and the web-service interface to return the heuristic types that are available for use by the planner.

3.4.3.3 Planner Interaction

Planner interaction in Bowyer is accomplished through the *Plan* menu. Before this functionality can be used the planning setup in the *Planning Setup* interface, selection of the planning domain and planning problem as described above, must be completed.. Bowyer's options for planner interaction include: allowing the user to open an existing plan saved in an XML Plan Node document, get the first complete plan from the planner, return the first N plans found by the planner or search for a given time limit and return all plans found by the planner.

Bowyer allows the user to open an existing plan from file that can then be used as if it were returned from a planner dynamically. The remaining three options for planning are variations of the way in which plans are retrieved from the planner. By selecting the "Get Next Complete Plan" item the planner will continue to search until the first complete plan is found and that plan node will be returned. By selecting the "Get Next (N) Plans" the planner will continue to search until *N* complete plan nodes are found by the planner. Finally, by selecting "Get Plans (N seconds)" the planner will search for *N* seconds and all complete plan nodes that are found by that time are returned. Using this final option does not guarantee any plans will be returned by the planner if no complete plans are found within that time period.

3.4.3.4 Viewing the Plan

Once a plan has been returned by the planner it can be viewed using Bowyer's interface in two formats. A first view of the plan is provided by Bowyer in the *Plan XML* interface as a tree structure where the plan node id, steps, causal links, ordering links and flaws in the plan are separate branches in the tree and can be expanded and collapsed as needed, the returned plan can be saved to file in a plan node document from this view. An additional view of the plan is provider in graph form in the plan graph view of the *Plan Execution* interface. This graph view shows the steps, the steps preconditions, constraints and effects and the causal and ordering links and allows the user to navigate through the graph view.

3.4.4 Plan Execution

Bowyer provides functionality to test generated virtual environment action classes and the returned plans in the virtual environment. Bowyer allows the user to execute each step in a plan in the correct order or to execute a specific operator provided that the preconditions for that operator have been satisfied in the current virtual environment world state.

3.4.4.1 Plan and Individual Operator Execution

Before the plan can be executed in the virtual environment Bowyer must connect to the Zócalo Execution Manager to interact with the virtual environment being used. This connection is created in the *Planning Setup* interface by specifying the URL location of the Execution Manager being used. If the *Plan Step* setting is selected then the plan is loaded and the first step in the plan is loaded for execution. When the first step is loaded the

operator for this step is loaded and the operator's parameters and the world objects these parameters are bound to are loaded for execution. The plan step can then be executed in the virtual environment. The request for the plan step to be executed is then sent to the Zócalo Execution Manager and sent to the Zócalo game client to be executed by the game client in the virtual environment. The step, if it succeeds, can be viewed executing in the virtual environment. The next step in the plan is them automatically loaded and ready for execution.

Execution of outside operators can be accomplished by selection of the *Outside Step* setting. The user is then able to load a set of operators to be executed from a Domain document that specifies these operators. The user can then load virtual environment objects from a Set State and Manage Plan document and create the binding between operator parameters and world objects manually using the functionality provided in the *Plan Execution* interface. Once the bindings are complete the operator can then be executed in the virtual environment in the same way the plan step operator is executed, if the operator's preconditions are met in the current virtual environment world state the operator will execute and can be viewed in the virtual environment.

Bowyer also provides a utility to view the current status of the Execution Manager being used by Bowyer to execute the plan. The *Execution Manager Status Monitor* interface shows, in list form, the operators in the planning domain maintained by the Execution Manager, the operators in the outside planning domain used for individual operator execution, the user initiated actions, the environment initiated actions and the literals that specify the current environment state.

4 Bowyer Usage Example

This section walks through a sample usage of Bowyer using an example based around a simple Bank Story domain. The Bank Story domain consist of a bank level built in the Unreal game engine that represents a bank and the immediate vicinity around the bank, a set of operators in the bank world and a bank robber character. The goal of the plan is for the bank robber to have the gold from the bank vault at the end of the plan. This will require the bank robber to get the key to the vault, unlock the vault and retrieve the gold from the vault.

In the rest of this section I will present an example of each step in Bowyer's interface using the Bank Story scenario to aid in understanding. For brevity I will only specify one item per tab interface but the remaining Bank Story items are specified in a similar manner to the examples given. The entire Bank Story planning domain is used by the planner to generate the returned plan.

Before working with Bowyer the configuration information needs to be set so Bowyer can locate any existing data such as client information including code libraries, base class specification and to select the type of planner to use with Bowyer. Bowyer configuration is set by selecting the *Edit->Configuration* menu item. This will display the *Configuration Dialog*. The user needs to select the Bowyer installation directory and the type of planner to use with Bowyer; this example will use the internal planner.

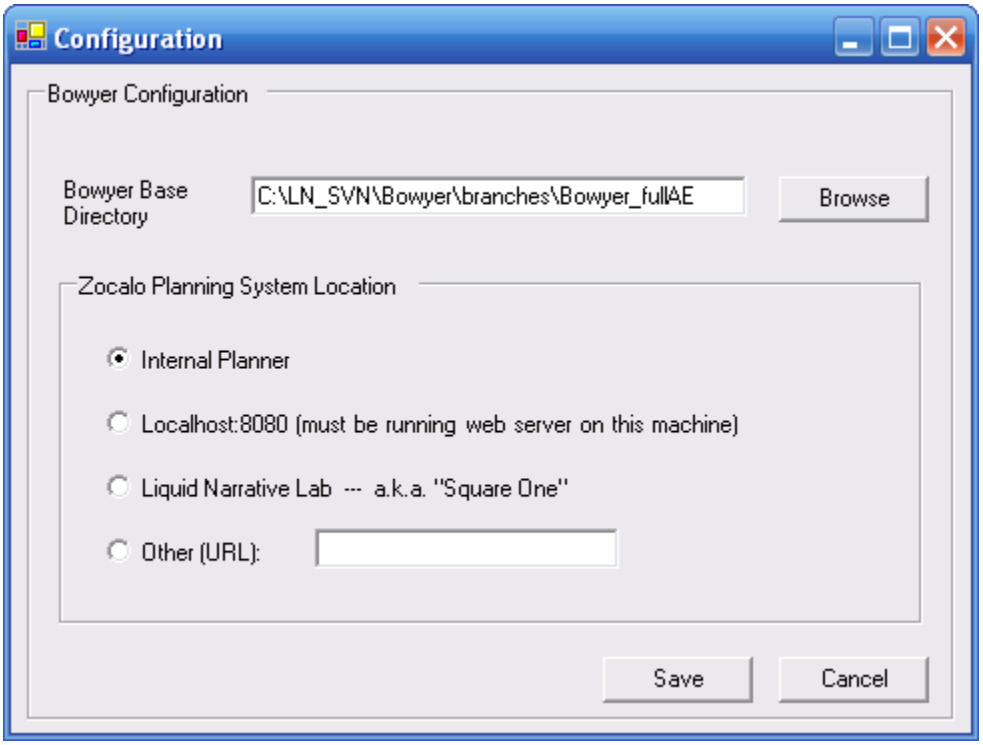

Figure 4.1 Configuration Dialog

4.1 Planning Domain Specification

The planning domain for use in the Bank Story scenario must be created by specifying the types, objects, literals and operators that will be used in the Bank Story domain.

4.1.1 Type Specification

The Bank Story has a type to represent the gold object in the Bank Story domain. This type is named “gold” and is a sub-type of the type “anyThing” which is the root of all types used in all domains specified by Bowyer. To create the “gold” type the user selects the “anyThing” type, enters “gold” into the *Type Name* field and selects an icon for this type

from the *Icon* pull down menu. Once the *Add Child* button is selected, the "gold" type will be displayed in the type tree as a child of the "anyThing" type.

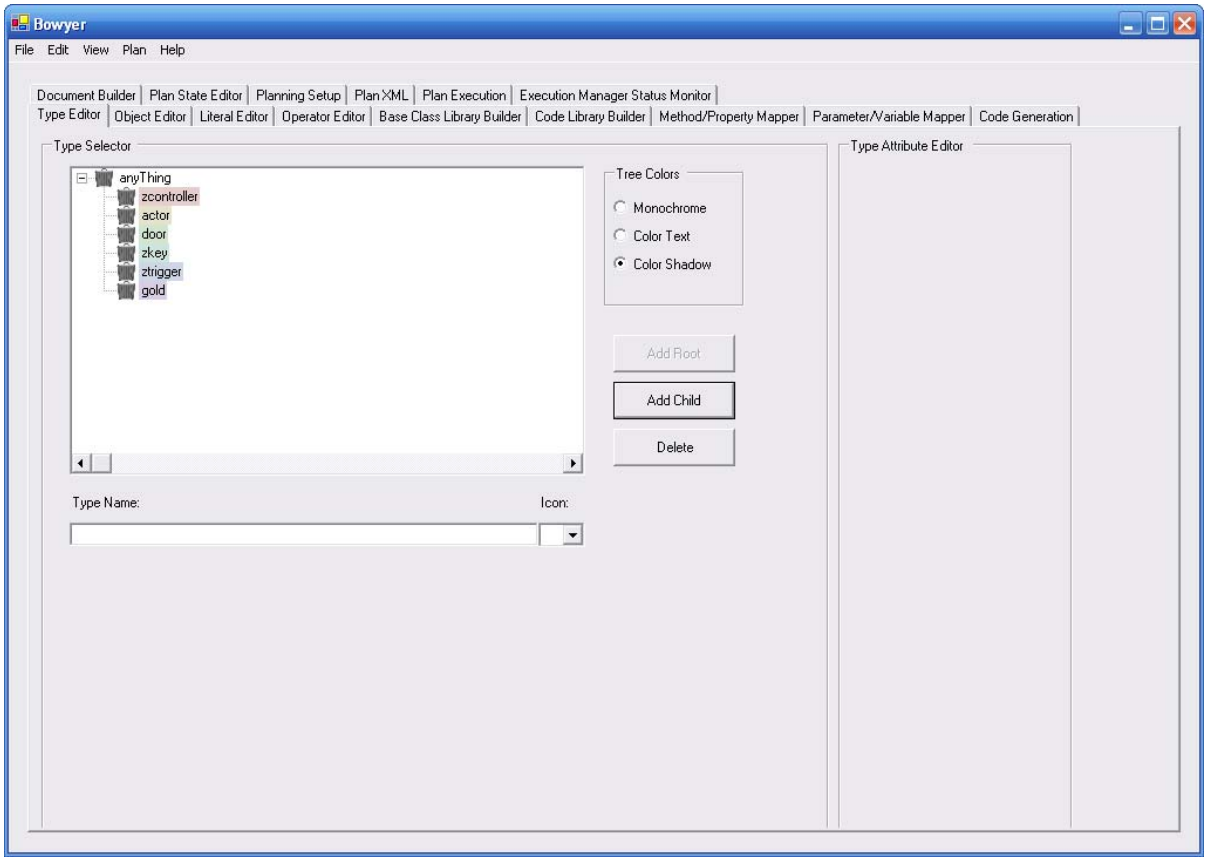

Figure 4.2 Type Editor

4.1.2 Object Specification

The objects used in the Bowyer planning domain specification represent the objects and locations found in the virtual environment world state. In the Bank Story domain one of the goal state conditions is that the bank robber, referred to as "ZPawn0", has the gold from the bank vault. For this to be possible a "gold" object must be created in the domain for the bank robber to have that object at the end of the plan. The gold object can be created in Bowyer in the *Object Editor* interface.

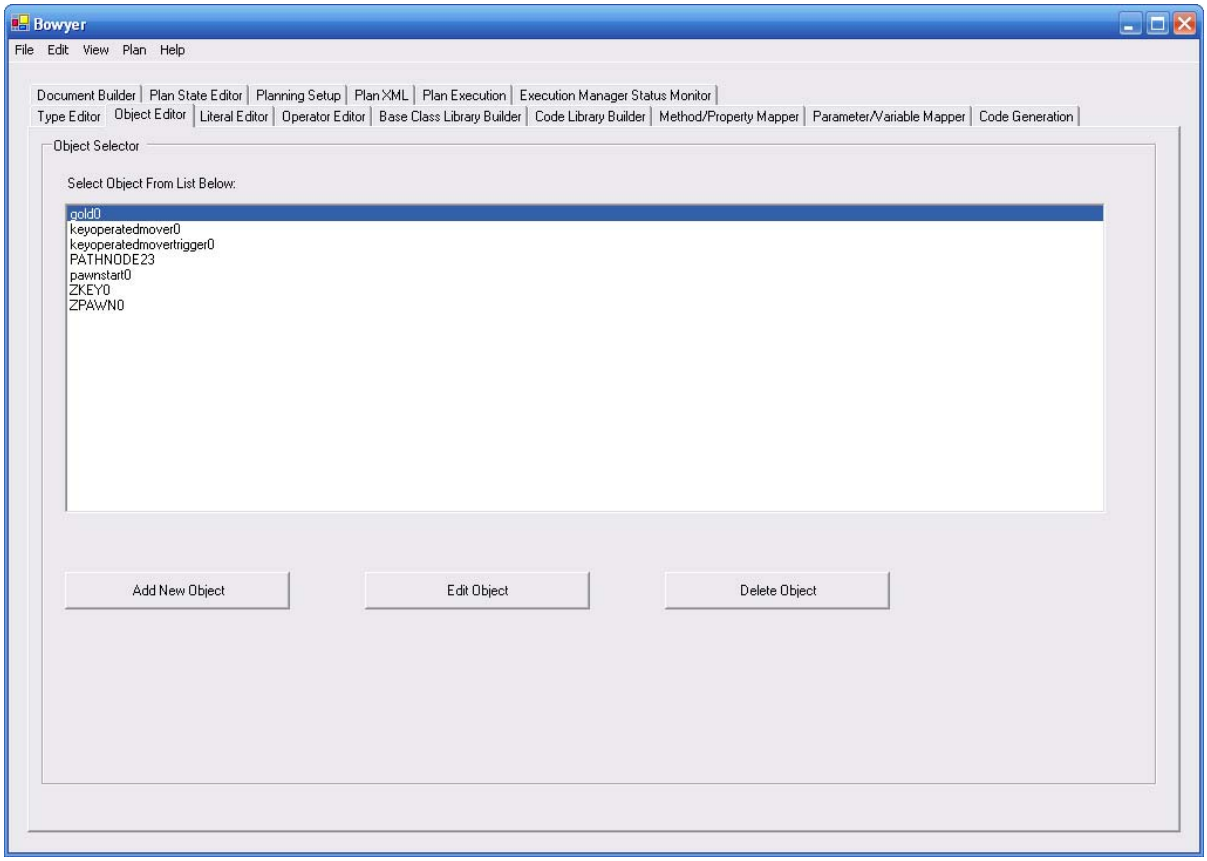

Figure 4.3 Object Editor

To create the gold object the user should select the *Add New Object* button which will display the *Object Edit Dialog* interface to the user to specify the properties of the gold object to be used in the Bank Story domain.

The only properties of the gold object that needs to be specified are the name "gold" and the type of the object. For the gold object the type is the "gold" type created in the previous step. The type gold is added by selecting "gold" from the type tree and selecting the *Add Type* button. The *Apply Changes and Exit* button can then be selected and the gold object will be added to the list of objects available in the planning domain.

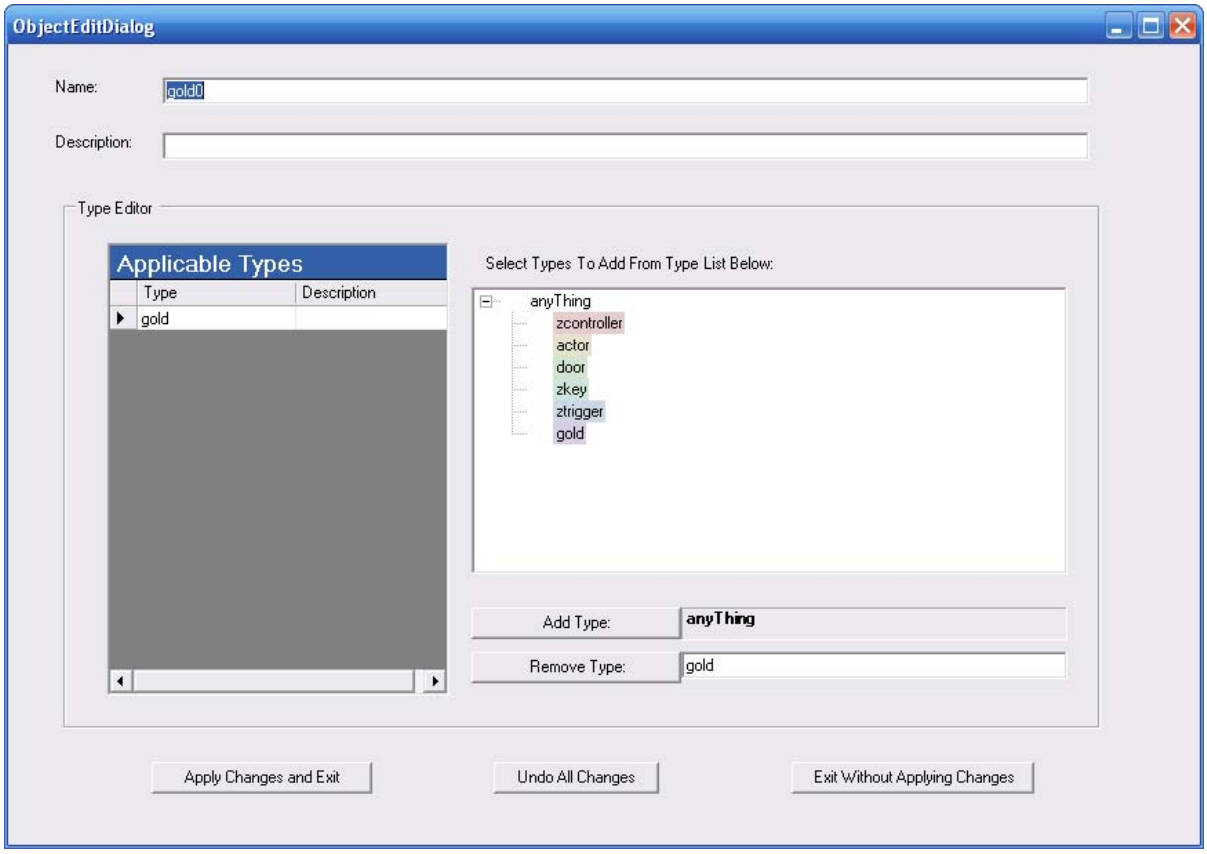

Figure 4.4 Object Edit Dialog

4.1.3 Literal Specification

Literals used in the planning domain specified in Bowyer represent all of the conditions that will need to hold true, false or be checked in the domain. One of the literals in the Bank Story that is used to specify a goal state condition is that the bank robber must have the gold at the end of the plan. The generic literal for "has" is specified by "has(zcontroller, anyThing)" in the *Literal Editor* used by Bowyer, to specify that an object of type "zcontroller" has any object of type "anyThing".

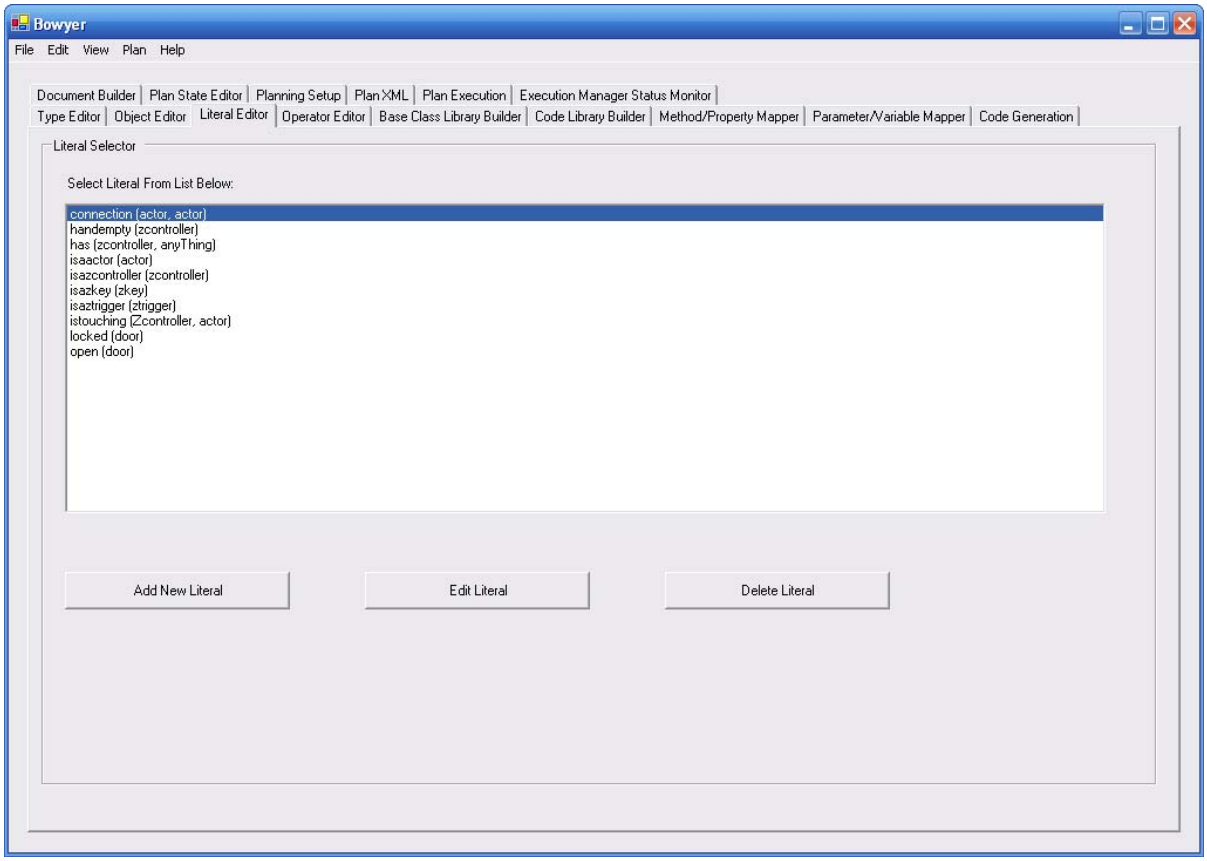

Figure 4.5 Literal Editor

Literals are specified in Bowyer very similar to the way Objects were specified in the previous step. The user must select the *Add New Literal* button which will display the *Literal Edit Dialog* to specify the Literals properties.

The only properties of this literal that needs to be specified for Bowyer's use is the literal name and the type of the arguments used by the literal. In the case of the "has(zcontroller, anyThing)" literal used by the Bank Story the type of the non player character is "zcontroller". The type "zcontroller" is added to the list of terms by selecting the "zcontroller" type from the type tree and selecting the *Add Type* button. The "anyThing"

type is added in the same way and the *Apply Changes and Exit* button is selected to save the literal to the planning domain for the Bank Story.

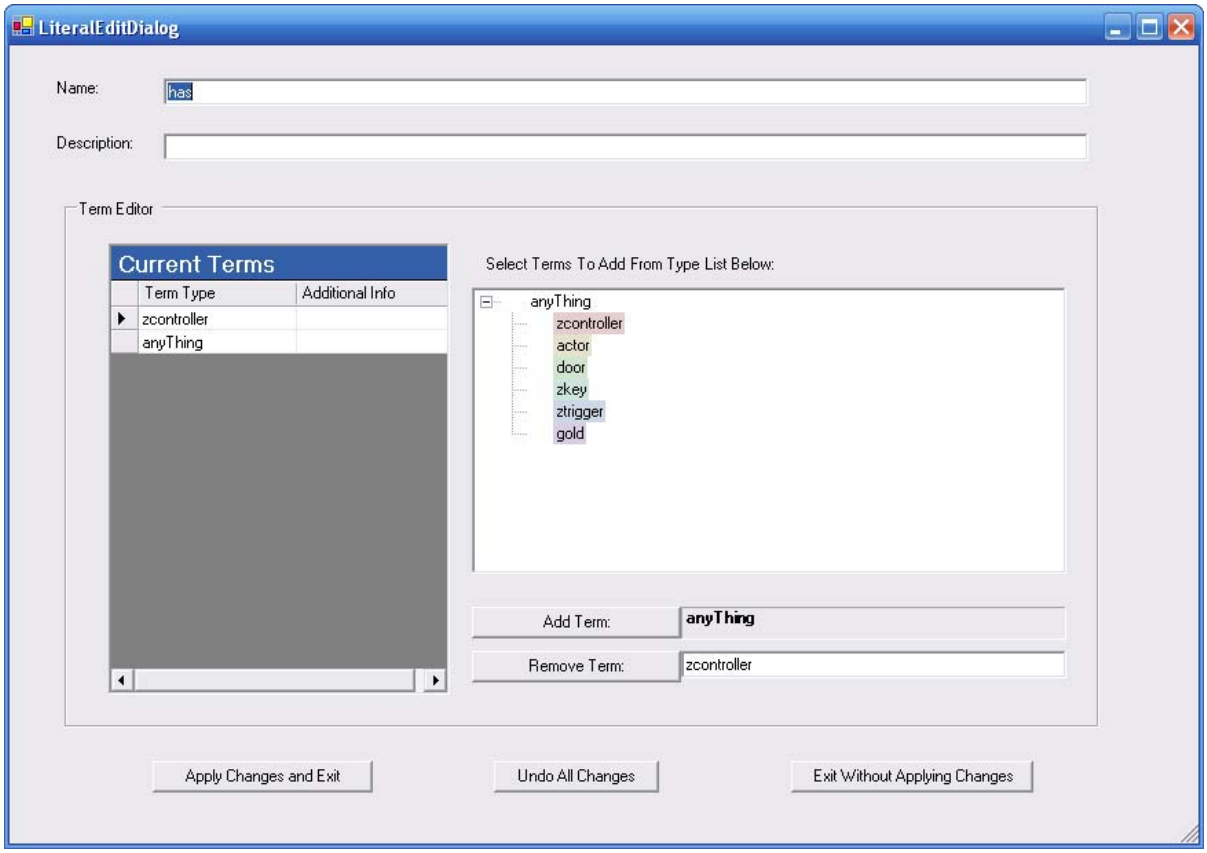

Figure 4.6 Literal Editor Dialog

4.1.4 Operator Specification

Operators in the Bowyer planning domain represent the possible actions in the planning domain that can be executed in the virtual environment. One of the operators used in the Bank Story domain is the "ZAction_ZController_MoveTo" operator. This operator represents that action of a non-player character moving from one location in the virtual environment to a different location in the virtual environment. The

“ZAction_ZController_MoveTo” takes three parameters: “?pawn”, “?target” and “?from”. The “?pawn” parameter represents the non-player character that will be moving in the virtual environment. The “?target” parameter represents the target location where the non-player character should be located when the operator has completed execution. Finally, the “?from” parameter represents the current location of the non-player character when the operator is going to be executed.

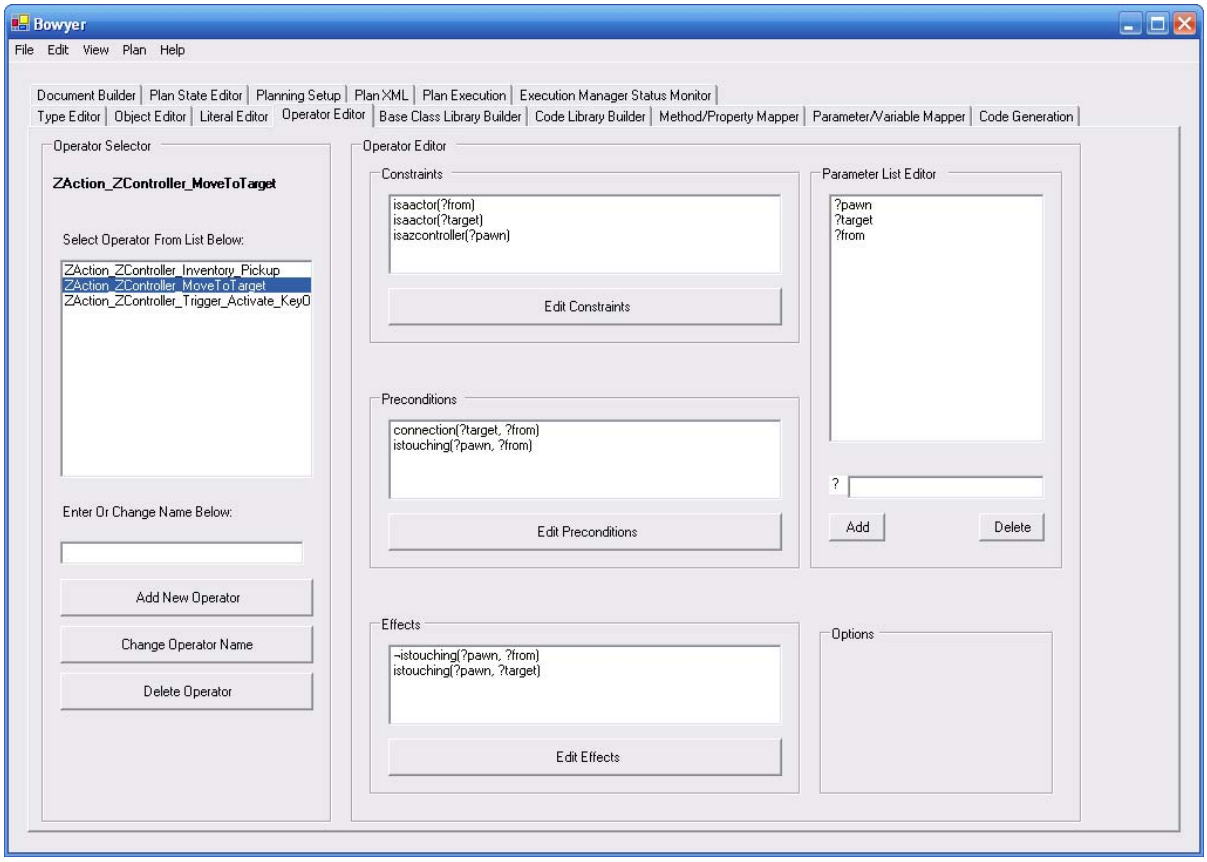

Figure 4.7 Operator Editor

The “ZAction_ZController_MoveTo” operator is specified in Bowyer by first entering the name “ZAction_ZController_MoveTo” in the operator name text box and

selecting the *Add New Operator* button which will add the operator to the list of operators in the domain. The three parameters are then entered by entering the name of the parameter in the parameter text box and selecting the *Add* button. This process in repeated for each parameter used by the operator. After the operator name and parameters have been specified the constraints, preconditions and effects for the operator can be specified.

Constraints are specified for an operator by selecting the *Edit Constraints* button which will display the *Operator Edit Component*. The set of literals that were defined in the *Literal Editor* are listed in the available literal list. The literals that define the constraints for the "ZAction_ZController_MoveTo" operator constraints are selected and the *Add* >>> button is selected to add the literal to the active literal list. Once the literals are in the active literals list each literal is selected and the arguments used in the literals are then mapped to the operator's parameters. For the "ZAction_ZController_MoveTo" operator the first constraint literal to add is "isaactor(actor)" the "actor" term is then mapped to the "ZAction_ZController_MoveTo" parameter "?from". The mapping is created by dragging "?from" into the empty box next to "actor" in the *Literal Adaptor* box. The constraints for the "ZAction_ZController_MoveTo" operator are that the "?from" and "?target" location must both be of the "actor" type and that the "?pawn" non-player character must be of type "zcontroller".

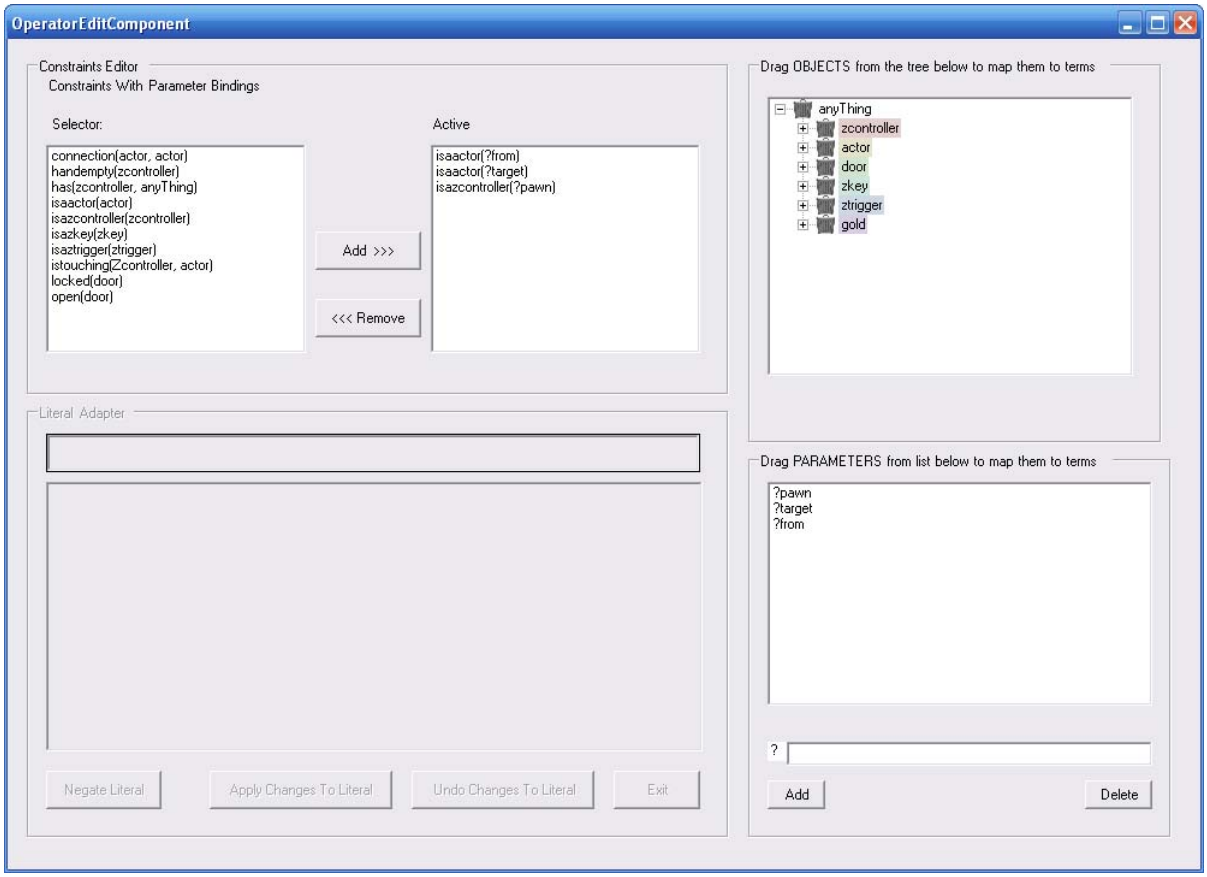

Figure 4.8 Constraint Edit Component

Operator preconditions and effects are specified is a similar process to operator constraints. The preconditions for the "ZAction_ZController_MoveTo" operator are that there is a path connecting the current location and the target location and that the non-player character is still located at the current location.

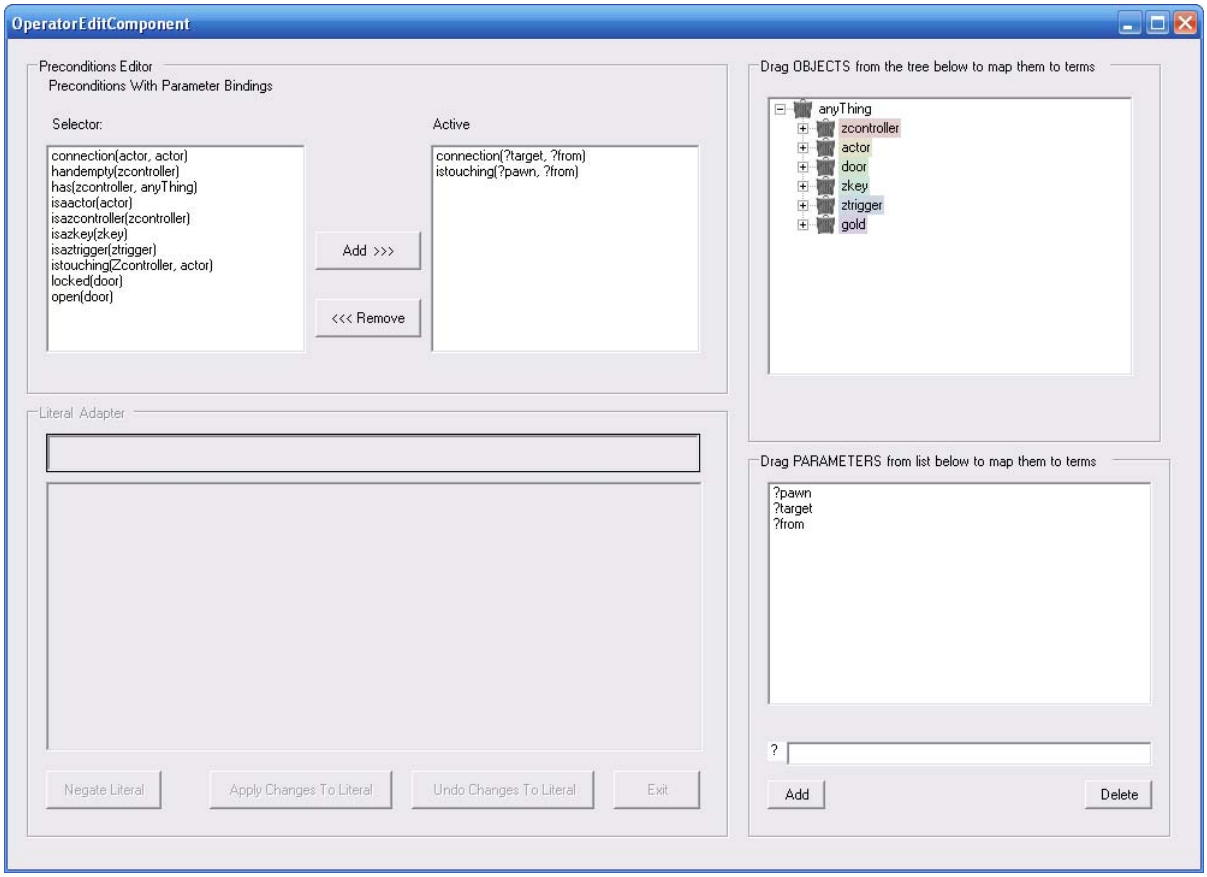

Figure 4.9 Precondition Edit Component

The "ZAction_ZController_MoveTo" operator's effects are that the non-player character is no longer touching the location that the non-player character was initially at before the operator was executed and that the player is now located at the target location at the end of the execution of the operator.

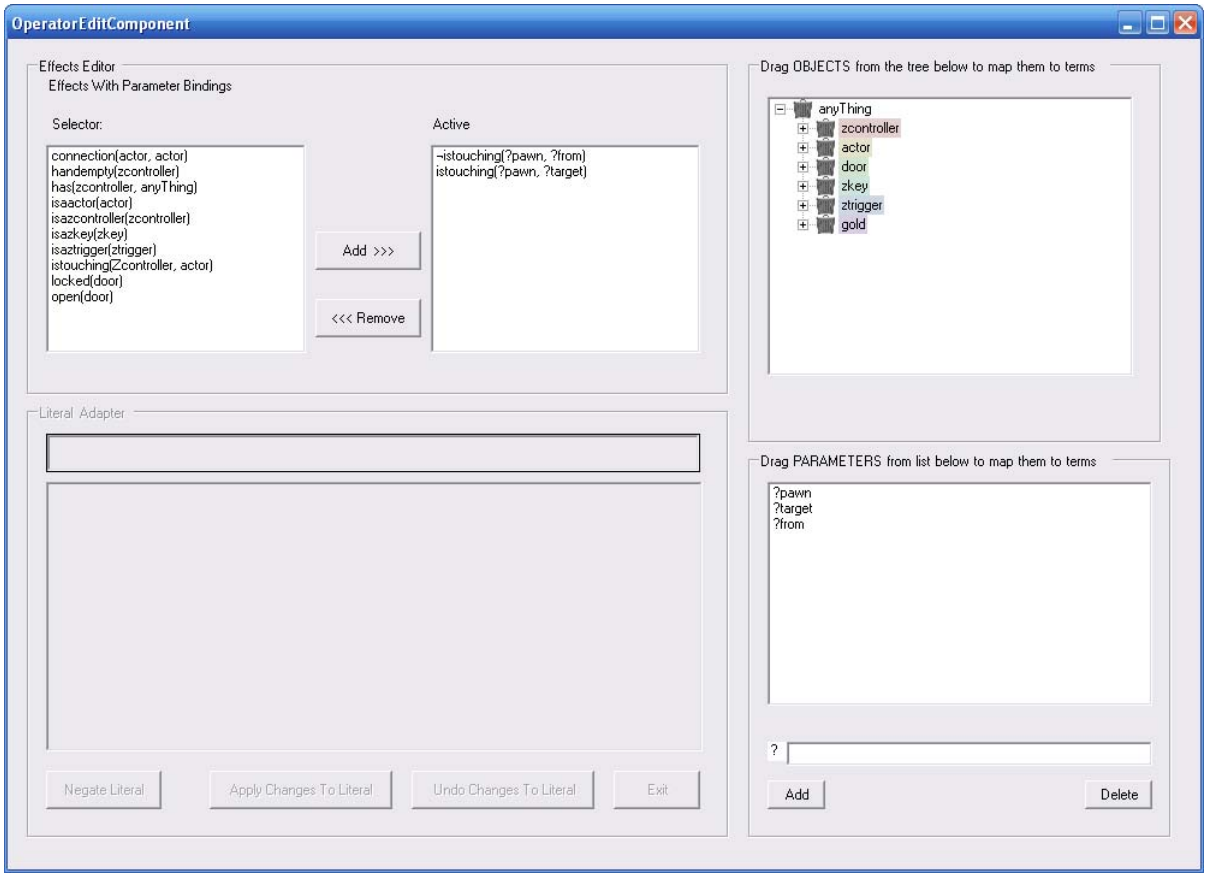

Figure 4.10 Effect Edit Component

This completes the specification of the "ZAction_ZController_MoveTo" operator in the Bank Story domain.

4.2 Client Specification

The client specification for the Zócalo game client used in the Unreal game engine must be created as well as any source code modules that need to be added to the procedural representation of the operators and objects. The base class specification only needs to be specified once per game client and the code library for the game client can be built overtime and reused.

4.2.1 Base Class Specification

The *Base Class Library Builder* interface in Bowyer is used to specify the generic operators and objects used in the Zócalo game client for the virtual environment in which the plan will be executed.

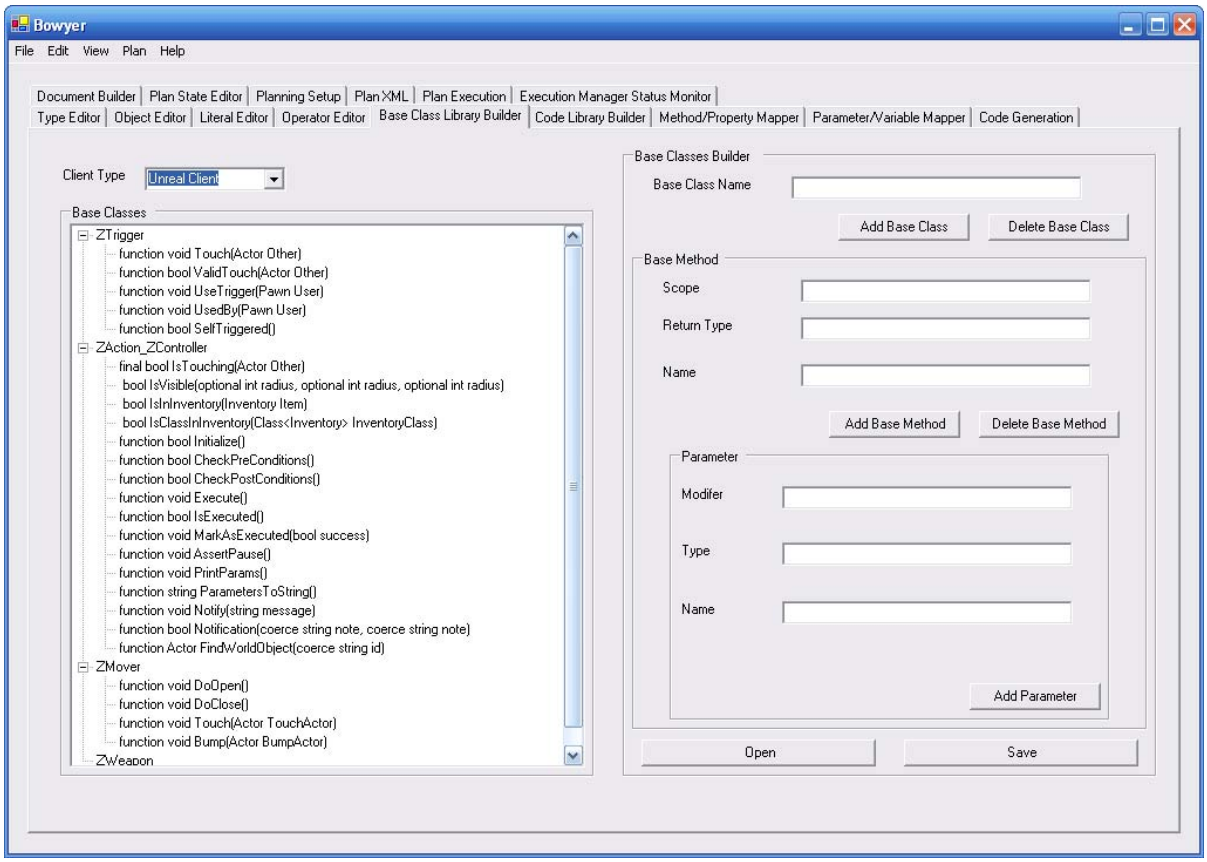

Figure 4.11 Base Class Library Builder

The first step in specifying a base class representation is to select the client type from the *Client Type* drop down menu that the base class representation will be created for. To specify the base class "ZTrigger" in the Bowyer interface, first the base class name "ZTrigger" must be entered into the *Base Class Name* text box and select the *Add Base Class* button. Once the base class name has been added the methods in the base class can then be

added. The first method "Touch" of "ZTrigger" is added to the base class by entering the scope "function", the return type "void" and the name "Touch" for the method and selecting the *Add Base Method* button. Once a method has been added the parameters that are used by that method can then be added to the method. The parameter in the method "Touch" is added by entering the type of the parameter "Actor" and the name of the parameter "Other", and selecting the *Add Parameter* button, this parameter does not have a modifier so the *Modifier* text box can be left blank. Each parameter per method is added in this way and the order of the parameters must match the base class used in the Zócalo game client. The additional methods for the "ZTrigger" base class and additional base classes in the Zócalo game client are added in the same way specified above until all of the client's base classes have been defined completely.

4.2.2 Code Library Building

The *Code Library Builder* interface allows the user to add methods and properties to the code library for each client to be used later in the mapping phase to map to the planning operator and object attributes. In the Bank Story domain the "ZAction_ZController_MoveTo" operator needs to execute an operation that moves the non-player character it is referencing from its current location to the target location. This will require that code for the client that moves a non-player character around in the virtual environment world.

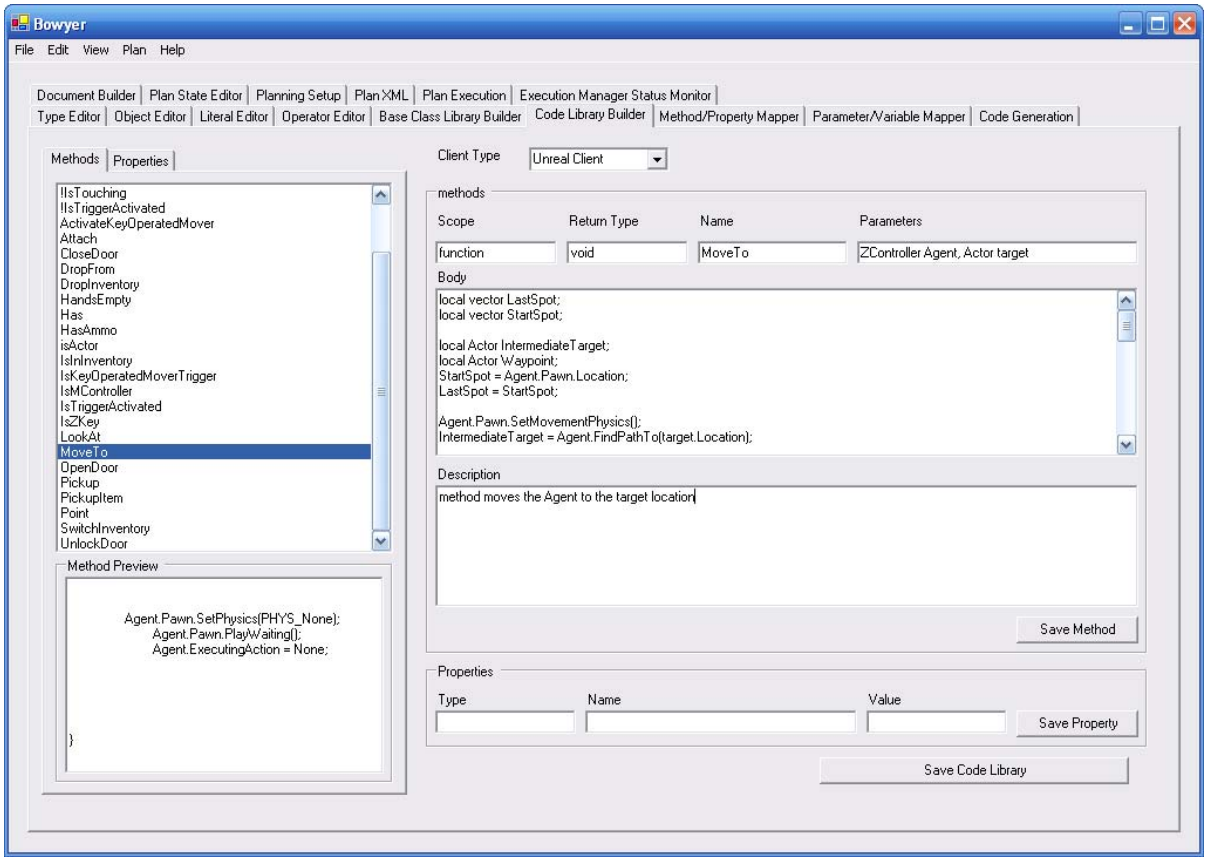

Figure 4.12 Code Library Builder

The first step in using the *Code Library Builder* interface is to select the client type that the inputted method will be used with from the *Client Type* drop down menu. This will load the current code library for that client so the user can see the methods and properties that already exist for that client. To add the "moveTo" method using the Bowyer interface the scope "function", return type "void", name "MoveTo" and parameters "ZController Agent, Actor target" need to be specified. The parameters are specified by entering the optional modifier, type and name of the parameter followed by a comma followed by the next parameter in the same manner it would be in the source code for that programming language.

The body of the method is then added to the *Body* text box and a description of what the method does is added in the *Description* text box. The description text will be displayed in the mapping interface to aid the user in the mapping process. When the *Save Method* button is selected the method is added to the methods in that client's code library. Properties for the code library are specified in a similar manner.

4.3 Code Generation

After the base classes for a client type have been specified and the code modules needed to generate the classes have been specified the mapping for methods and properties can be created and the mapping from the operator's parameters to the variables in the mapped methods code can be created. The generation of the procedural representations of the planning operators and objects can then be generated using Bowyer's provided functionality.

4.3.1 Method and Property Mapping

The source code methods and properties that represent the virtual environment equivalents to the planning attributes have to be mapped to the planning operators and objects by the user.

For the "ZAction_ZController_MoveTo" operator the Effect literal "~isTouching(?pawn, ?from)" needs to be mapped to the "!IsTouching" method. This is accomplished by first selecting the "ZAction_ZController_MoveTo" operator from the *Operators* tab. This will populate the preconditions, constraints and effects lists for that operator. The user then selects the operator attribute to be mapped, in this case the effect "~isTouching(?pawn, ?from)" effect is selected. This sets the planning component of the

mapping to “Effect” specifying that you are mapping an effect, or postcondition, of the operator.

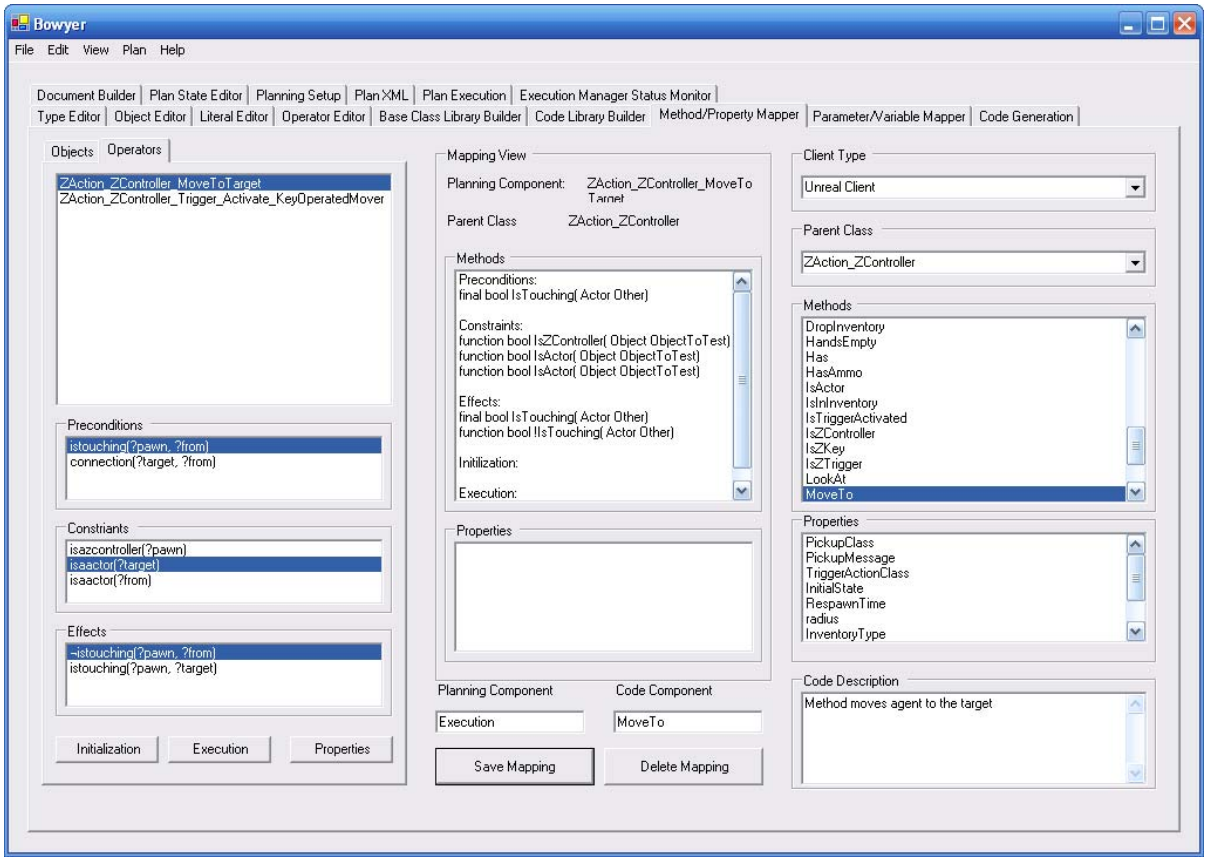

Figure 4.13 Method/Property Mapper

The code attribute to be mapped is selected by first selection the client type, “Unreal Client”, for which the mapping will be used from the *Client Type* pull down menu. This will populate the *Parent Class* pull down menu that contains the base classes specified for that client. The user then selects the base class that the operator will extend. For the “ZAction_ZController_MoveTo” operator, and all operators in the unreal environment, the base class is “ZAction_ZController”. Once the parent class is selected the parent class’s

methods and all of the other methods contained in the code library are loaded into the *Methods* list and all of the properties from the code library are loaded into the *Properties* list. The "!IsTouching" method is them selected which sets the *Code Component* in the mapping panel to "!IsTouching". The *Save Mapping* button is then selected and the "!IsTouching" method is added to the mapping view in the methods list under the *Effects* heading. The remaining methods are mapped in a similar manner. Any methods that need to be used for initialization of the operator or execution of the operator are mapped by selecting the appropriate button instead of a planning attribute from a planning attribute list. For the "ZAction_ZController_MoveTo" operator the *Execution* button is selected for the planning component and the "MoveTo" method is selected from the methods list for mapping. Methods and properties are mapped to planning objects is the same manner methods are mapped to operators.

4.3.2 Parameter to Variable Mapping

Once all of the methods for an operator are mapped to that operator the next step in the code generation process is to map the operator's parameters to the variables used in mapped methods' parameters. To avoid confusion the methods' parameters are referred to as variables.

For the "ZAction_ZController_MoveTo" operator the "IsTouching(Actor Other)" method is mapped to a precondition and an effect, the "!IsTouching(Actor Other)" method is mapped to an effect, the "IsActor(Object ObjectToTest)" and "IsZController(Object ObjectToTest)" methods are mapped to constraints and the "MoveTo(ZController Agent, Actor target)" method is mapped to the execution of the operator. This mapping is what

populates the parameter to variable mapping panel. The variable "Other" in the precondition "IsTouching(Actor Other)" is mapped to the operators "from" parameter, the variable "Other" in the effect "IsTouching(Actor Other)" is mapped to the "target" parameter

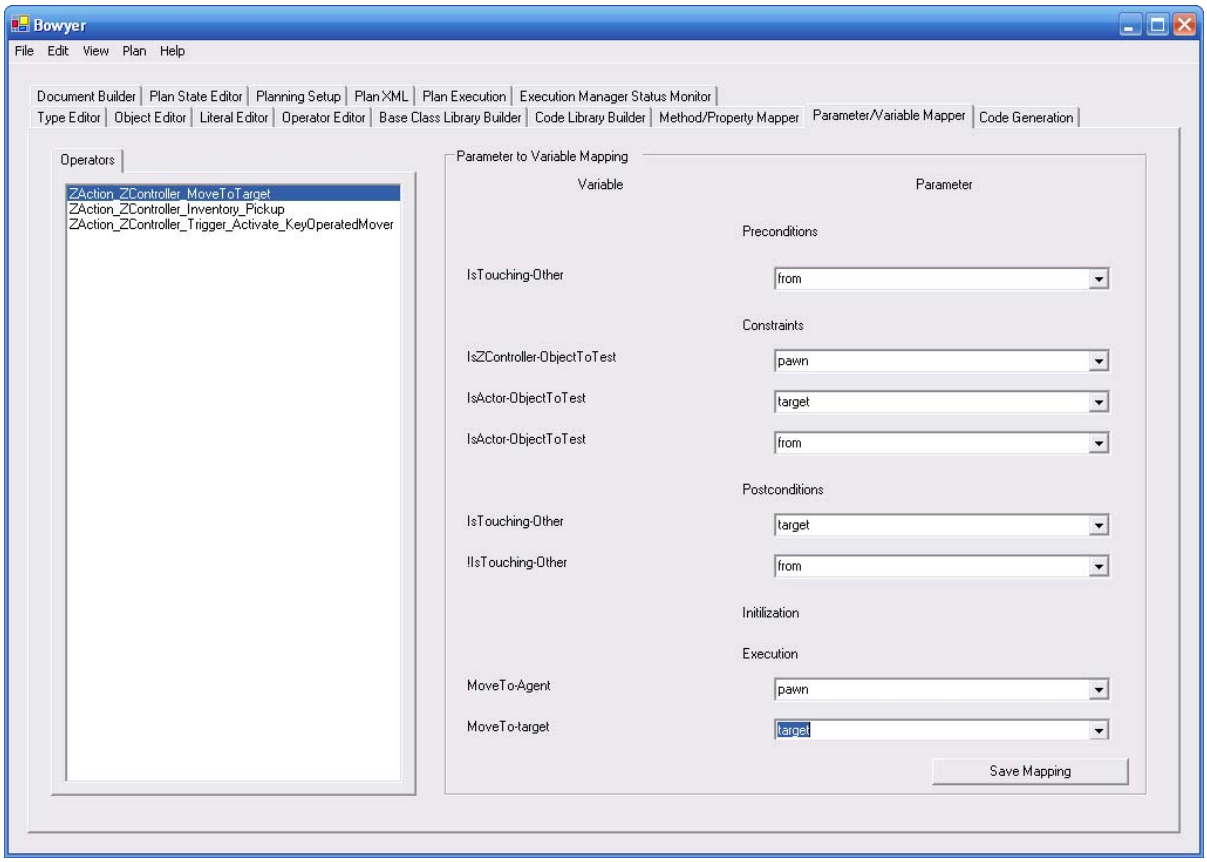

Figure 4.14 Parameter/Variable Mapper

and the variable "Other" in the effect "!IsTouching(Actor Other)" is mapped to the "from" parameter. This represents that the non-player character is touching its current location before the operator is executed and touching the target location after the operator is executed and thus no longer touching the location it was before the operator executed. The constraints "IsActor(Obejct ObjecToTest)" is mapped to "from" and "target" and "IsZController(Object

ObjectToTest)" is mapped to "pawn" this represents that the location that the non-player character is moving from and the location that the non-player character is moving to must be of type "Actor", a convention in Unreal Script, and the non-player character must be of type "zcontroller". The "MoveTo(ZController Agent, Actor target)" "Agent" variable is mapped to the "pawn" parameter and the "target" variable is mapped to the "target" parameter representing that the non-player character is moving from it's current location to the target location during the execution of the operator. Once all of the mappings have been assigned the mappings are saved to be used in code generation in the next step.

4.3.3 Code Generation

Once all of the mapping is complete, code generation for the operators and objects is straight forward. The user selects the client type to generate for, "Unreal Client" from the *Client Type* pull down menu. This will populate the *Objects* and *Operators* list for that client. The "ZAction_ZController_MoveTo" operator is then selected and a name to use for the generated action class, such as "MoveTo", is entered into the *Class Name* text box. The *Generate Code* button is then selected and the generated code is displayed in the *Generated Code* text box. The generated code can then be changed and formatted if needed and saved to file by selecting the *Save* button.

Once the *Generate Code* button is selected an XML specification of the code to be generate is created and saved to the location "[*Bowyer Base Directory*]\Generated Code\[*Client Type*]\XMLFiles\[*File name*].xml". An example of the generated XML specification for "ZAction_ZController_MoveTo" is available in appendix 8.2. After the XML specification is generated it is sent to an XSLT processor along with the XSLT

templates for the client type that is being used. This generates the source code representation of the operator and this resulting code is displayed in the *Generated Code* text box. Once the

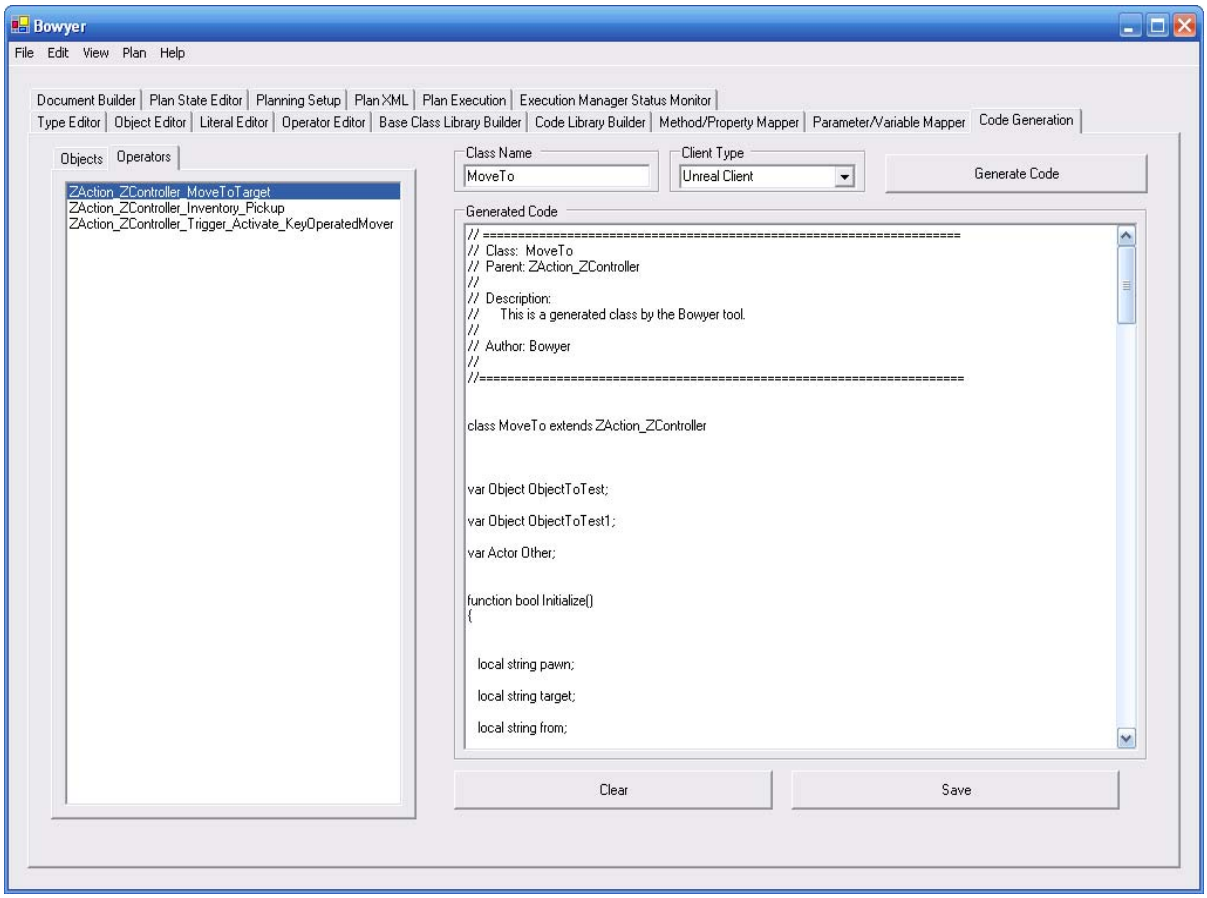

Figure 4.15 Code Generation Editor

Save Button is selected the source code is saved to the location "[*Bowyer Base Directory*]\Generated Code\[*Client Type*]\CodeFiles\[*File name*].[*Client Type Extension*]".

4.4 Document Specification

To use Bowyer and the Zócalo framework to execute a plan the minimal XML documents that must be created is the Set State and Manage Plan document that is associated

with the virtual environment to use for execution. Bowyer also allows the user to specify the other XML documents that can be used by the Zócalo framework. The planning domain and problem to be used can be specified using the Bowyer interface. The image below shows the specification of the Set State and Manage Plan document for the Bank Story.

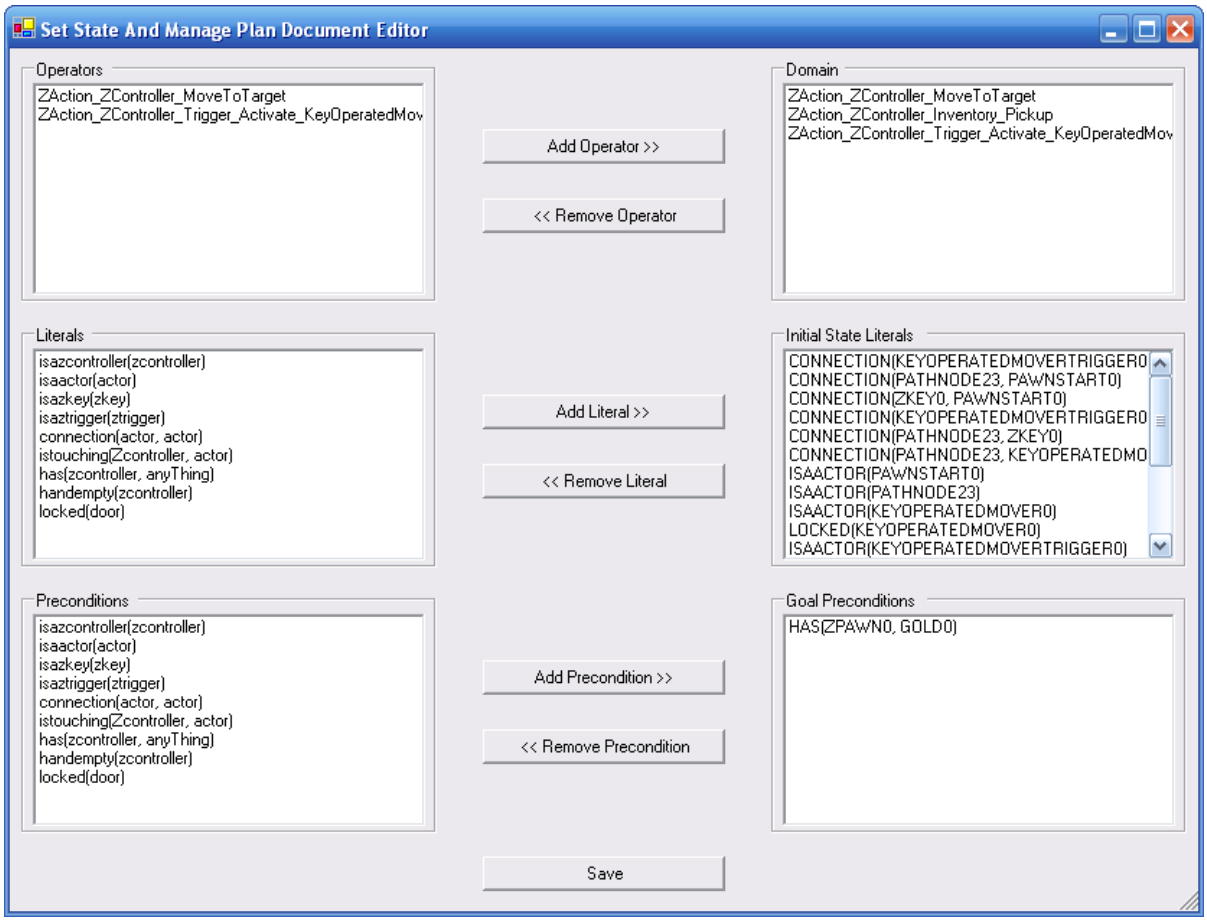

Figure 4.16 Set State and Manage Plan Document Editor

This interface allows the user to specify the operators in the domain, the initial state literals that specify the initial state of the world and the goal literals that specify the goal state for the plan. For literals that need to be bounded to world state object, such as in the initial

state and goal state, when the *Add Literal* button is selected a dialog to create the bindings is displayed to the user.

4.5 Plan Generation and Execution

Next the planning setup for the Bank Story will be specified including the planning domain and planning problem specification as well as selection of a search heuristic for use by the planner.

4.5.1 Plan State Specification

The Plan State Editor allows the user to specify the initial state of the world and the desired goal state of the plan. The initial state of the Bank Story specifies all of the facts and conditions that are true at the creation of the Bank Story world. These facts include the types of the objects in the world, the characters in the world, the objects the characters have in their possession and other properties of the world's characters and objects. Bowyer works under the closed world assumption. The goal state of the plan specifies the conditions that are desired at the end of the plan which for the Bank Story is that the bank robber has the gold from the bank vault.

To add literals to the initial and goal states the *Edit Initial State* or *Edit Goal State* buttons are selected and the same interface used to edit an operator's constraints, preconditions or effects is displayed to the user. Literals are added to the initial and goal states in the same manner literals are added to the constraints, preconditions and effects.

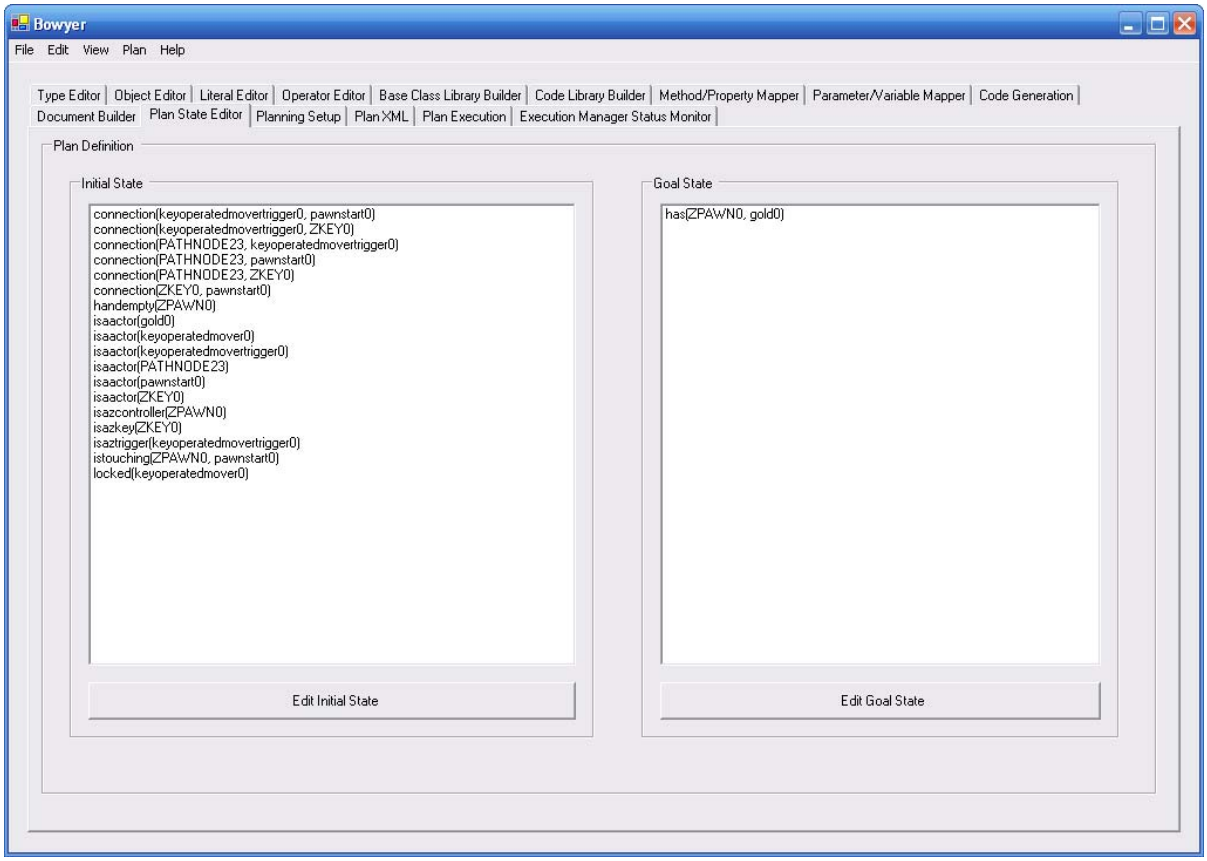

Figure 4.17 Plan State Editor

4.5.2 Planning Setup

After the planning problem has been created in the previous step the planning setup can be completed which consist of specifying the planner to use, planning domain to use, the planning problem to use, the starting plan node for the plan search and the heuristic to use for search in the planning process.

For this example the execution of the Bank Story is using the internal Bowyer planner, the planning domain to be used will be the planning domain maintained by Bowyer and the planning problem to be used is the problem that was specified in the *Plan State*

Editor in the previous step. The initial plan node is created using the planning problem specifications and will be using a heuristic to find the shortest plan possible.

The connection to the Zócalo Execution Manager is also established using the *Planning Setup* interface. The Execution Manager that is used in this example is running on the local machine and is located at the URI: "tcp://localhost:8988/UT2k4test".

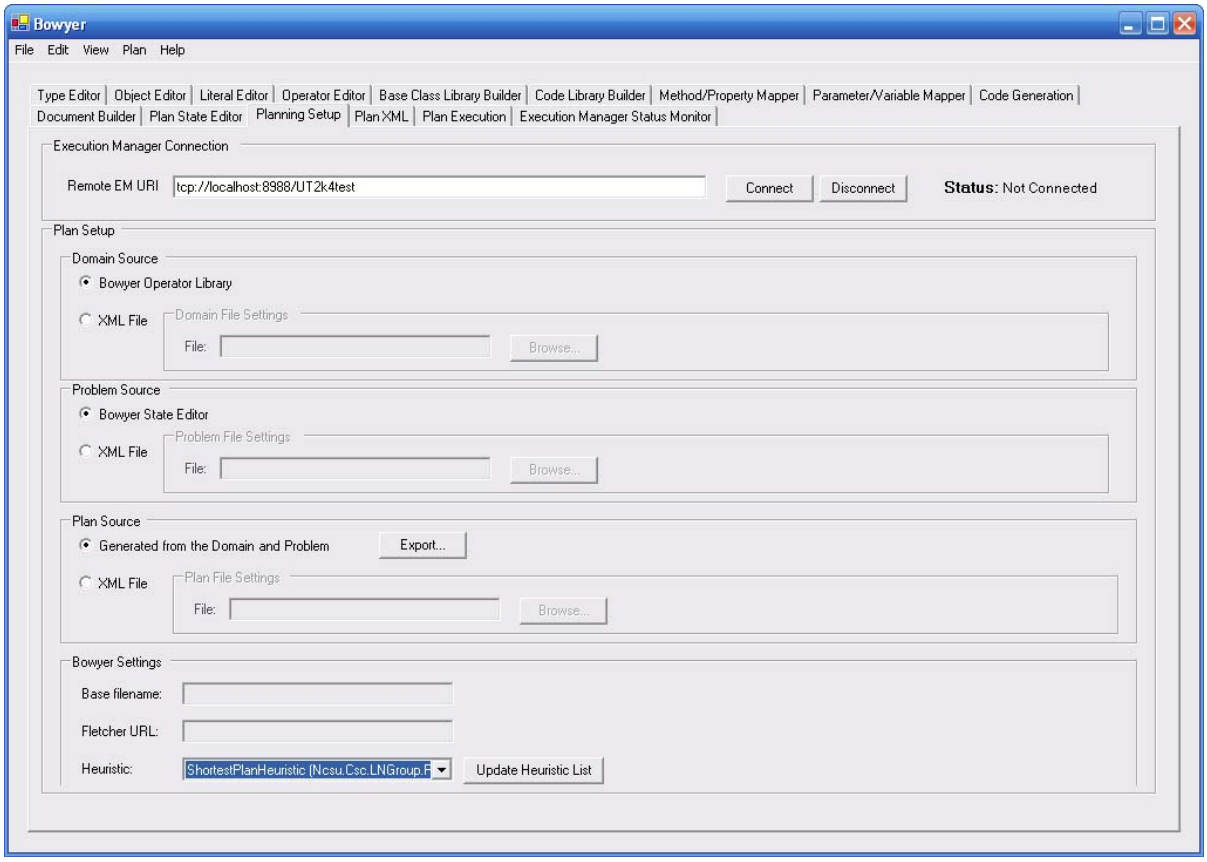

Figure 4.18 Planning Setup

4.5.3 Plan XML View

After the planning domain and planning problem have been specified the *Plan* menu can be used to get a plan from the planner using planning setup information selected previously. In this example the plan menu option *Get Next Complete Plan* is used to retrieve

the first plan found by the planner. Once the plan is returned from the planner, Bowyer has an XML representation of the plan that can be viewed in the *Plan XML* interface. This interface presents the XML in a tree view to allow the user to view the plan in an organized manner. The tree view displays the plan node id, plan steps, causal links, ordering links and flaws in the plan and also allows the user to save the plan to file.

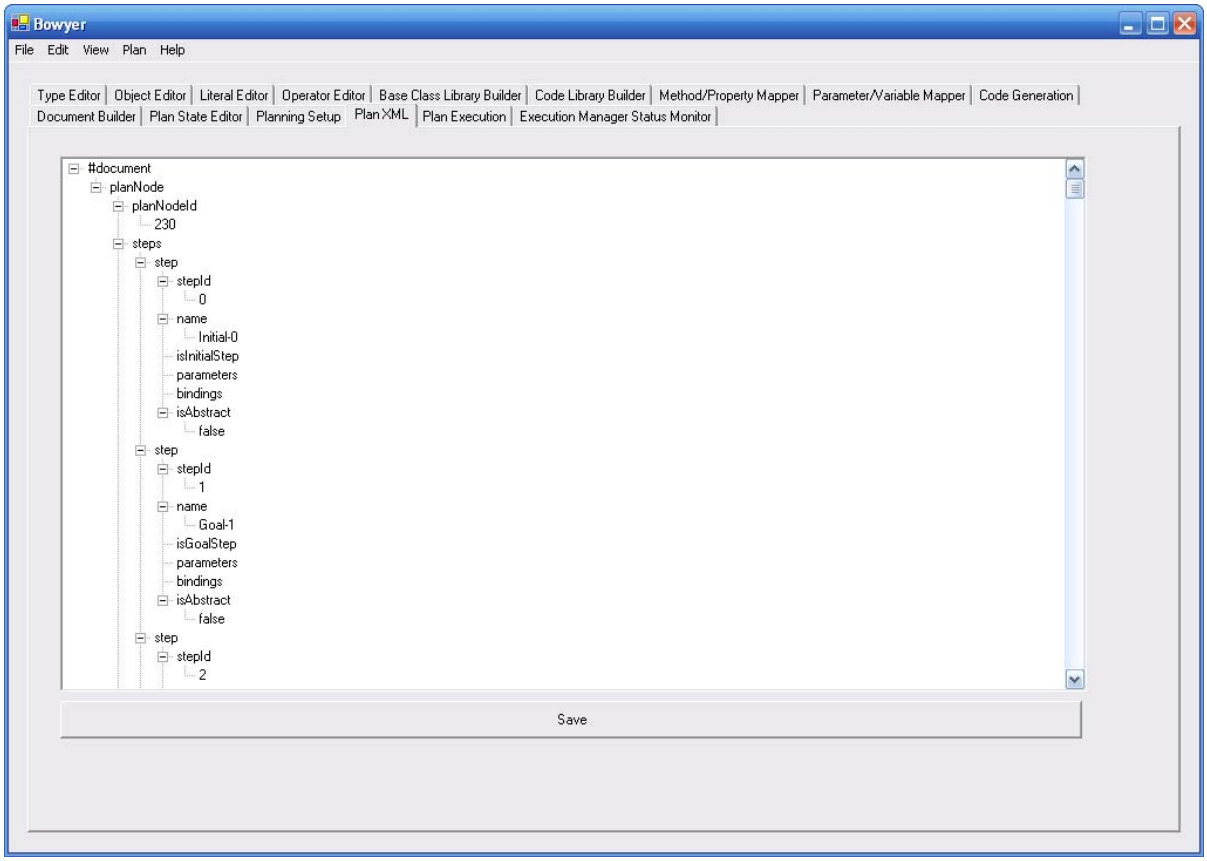

Figure 4.19 Plan XML

4.5.4 Plan Execution

The *Plan Execution* interface used in Bowyer displays the plan returned from the planner in a graph view display. Bowyer allows the user to execute action steps in the virtual

environment in two ways: execution of the steps in the plan returned by the planner or execution of individual operator steps for testing purposes.

Plan steps are executed by selecting the *Plan Step* radio button with will automatically load the first step in the plan. Loading the first step loads the operator that corresponds to the first step in the plan. This consists of loading the operator name and the operator's parameters along with the virtual environment objects that are bound to the operator's parameters. All of this information is retrieved from the plan specification that is

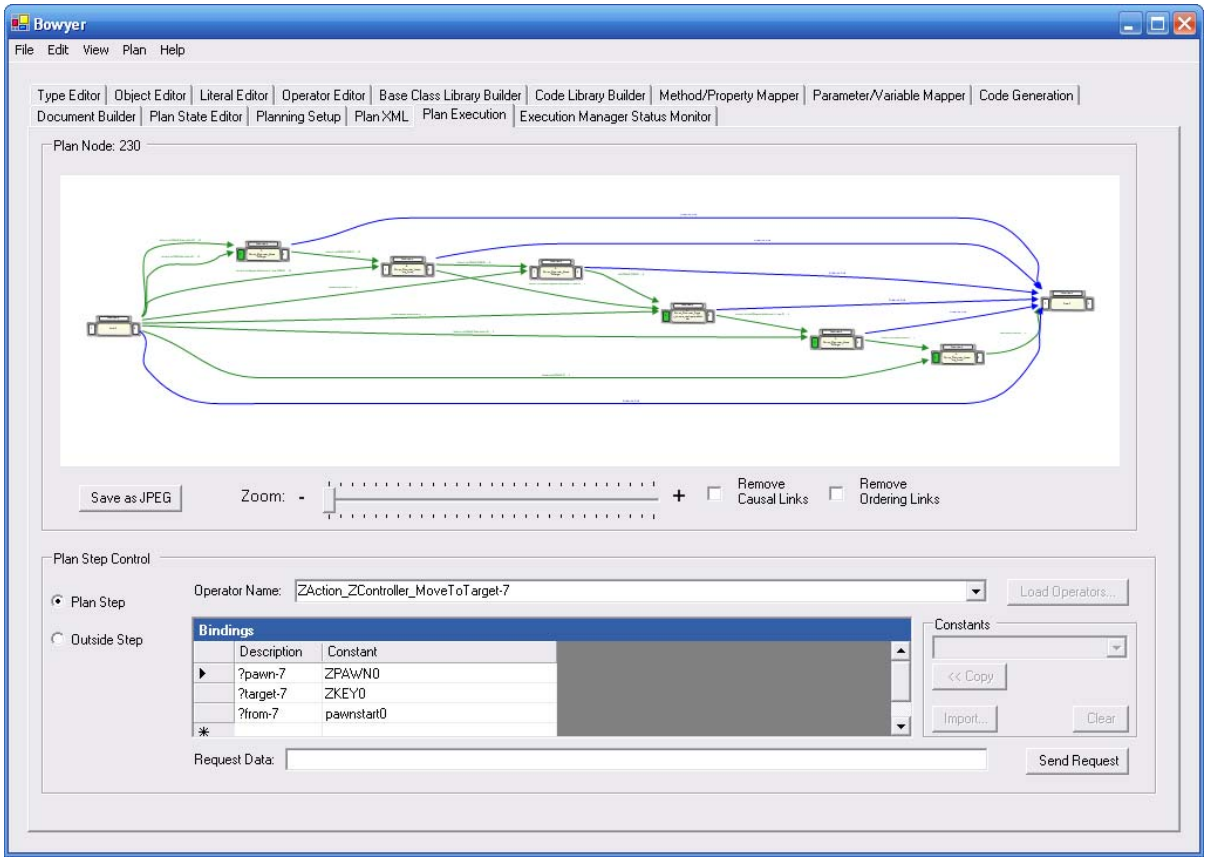

Figure 4.20 Plan Execution (Plan Step)

returned from the planner. For the plan returned for the Bank Story domain the first step in the plan is a "ZAction_ZController_MoveTo" operator where "?pawn" parameter is bound to "ZPawn0" which represents the bank robber non-player character in the Bank Story domain. The "?target" parameter is bound to the "ZKey0" Bank Story object which represents the location of the key to the bank vault that the robber needs to open the vault. Finally the "?from" parameter is bound to the "pawnstart0" object which represents the location of the bank robber character in the Bank Story world at the beginning of execution of the virtual environment. The "ZAction_ZController_MoveTo" operator represents the bank robber going to get the key to the bank vault as the first step in the plan for the bank robber to have the gold from the bank vault at the end of the plan. This first step is executed by selecting the *Send Request* button to send the step request to the Zócalo Execution Manager which sends the request to the Zócalo game client to be executed in the virtual environment.

The "ZAction_ZController_MoveTo" operator that is the first step in the plan returned for the Bank Story can also be executed outside of the plan by selecting the *Outside Step* radio button which will enable the *Load Operators*, << *Copy*, *Import* and *Clear* buttons in the *Plan Step Control* interface. To load the possible operators to be executed as an operator outside of the plan the *Load Operators* button is selected and a file browser allows the user to select a Domain document that contains the specification of the operators to be executed. This will load all of the operators from the selected domain document into the operators pull down menu.

The "ZAction_ZController_MoveTo" operator can then be selected from the pull down menu and the "?pawn", "?target" and "?from" parameters for that operator are loaded

into the *Bindings* table. To create the bindings from the operator's parameters to the virtual environment world objects first the virtual environment world objects need to be loaded into the *Plan Step Control*. This is accomplished by selecting the *Import* button and selecting the Set State and Manage Plan document that is associated with the Bank Story virtual environment being used. This will load the virtual environment's world objects into the *Constants* pull down menu. The bindings are then created by selecting the operator parameter, selecting the corresponding world object from the *Constants* pull down menu and selecting the <<< *Copy* button to create the binding.

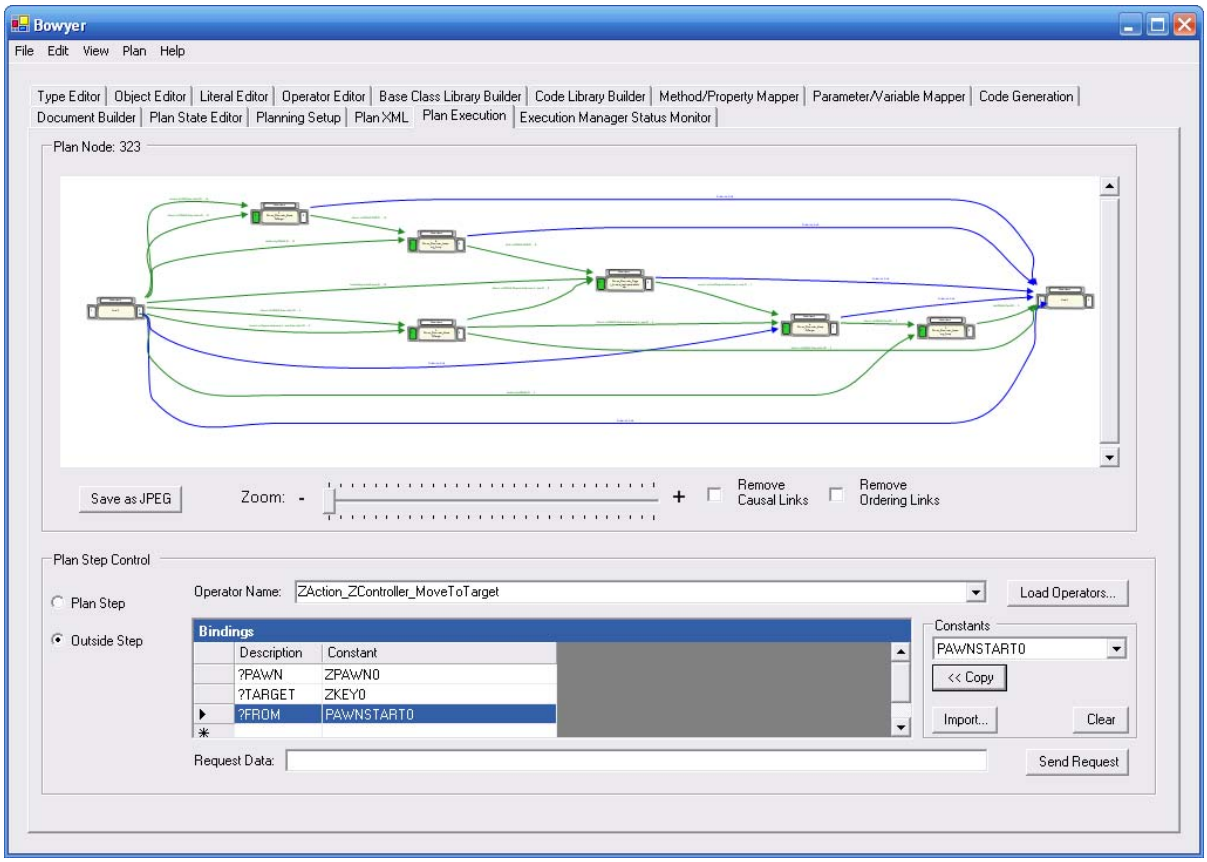

Figure 4.21 Plan Execution (Outside Step)

Once all of the bindings have been specified the step can then be sent to the Zócalo Execution Manager by selection the *Send Request* button. The step is then executed as if it were a step in a plan assuming the operator's preconditions are valid in the current world state.

5 Pilot Study

An initial pilot study for Bowyer was conducted to determine target users evaluation of the functionality Bowyer provides, its applicability to the tasks for which it was designed and its applicability to be used as a planning research tool. This section is broken into: an overview of the study providing context and goals for the study, an overview of the study's structure, the study findings and a discussion of the study's findings including possible reasons for the findings and future plans to address the findings in terms of changes to functionality, user interface and documentation provided with Bowyer.

5.1 Study Overview

Given that Bowyer is a proof of concept tool and in an alpha phase of development this study was used to determine the initial reaction of potential users after they have been given the chance to interact with Bowyer in a sample exercise to understand its functionality and scope. Accordingly this study's tasks were aimed at allowing the user to complete a set of sample tasks while restricting the task enough that the users were not overwhelmed, wer able to complete the study in a reasonable time frame and focus on the functionality provided by Bowyer more than completing the exercise.

This allowed the study participants to get an overall feel for the way Bowyer was designed to be used as well as the types of tasks that Bowyer could aid them in completing their planning research. This allows the participants to create an initial evaluation of the overall design of Bowyer and evaluate how well Bowyer fits into its designed role of aiding

planning researchers in integrating planning research into virtual environments as well as also evaluating the individual pieces of Bowyer's functionality.

5.2 Study Setup

The study was broken into three stages to correspond with the three distinct areas of Bowyer functionality. The first stage of the study was aimed at evaluating Bowyer's ability to allow the user to specify a complete planning domain using Bowyer's interface. The second stage of the study was used to evaluate Bowyer's ability to allow the user to generate virtual environment source code representations of the planning domain operators and objects for use in the planning domain. Finally, the third stage of the study was designed to evaluate Bowyer's ability to allow the user to interact with a planner and the Zócalo framework to create a planning problem specification, get a plan for the planner and execute the returned plan in the virtual environment using Bowyer's interface to accomplish these tasks.

Normally the use of Bowyer would be an iterative process where the user would work to create a planning domain and planning problem and attempt to find plans using the planning domain. This is largely a trial and error process and would require the user to add to and modify the planning domain and planning problem. This study is aimed at giving the user's an overall idea of the functionality provided by Bowyer. With this in mind, in order to make Bowyer's use fit into the structure and general time constraints of a study session, a partially specified planning domain and planning problem was loaded for the participants in the study and they were asked to complete the planning domain and planning problem using a textual description provided in the study instructions. The Unreal game client

configuration, Zócalo game client, Unreal base class specification and Unreal code library were also all provided for the users and are expected to be created by virtual environment developers and downloadable for use with Bowyer. Along with the study instructions the users were given a Bowyer User's Manual and a document that describes specific attributes of the virtual environment created by the Unreal game engine. A document of this type should be provided for each virtual environment that has an implementation of the Zócalo game client available.

The first stage of the study is used to evaluate Bowyer's ability to allow the user to specify a complete planning domain. The partial planning domain provided for the user is missing one type, literal, object and operator. The user was given a textual description of the type, literal, object and operator to be added and asked to use Bowyer's interface to enter these planning domain specifications.

The second stage of the study is used to evaluate Bowyer's ability to allow the user to create virtual environment representations of the planning domain objects and operators for use in executing a plan created using this domain in the virtual environment. In this stage the instructions the users were given contained an example mapping, method/property and parameter/variable mapping, and code generation for an object and operator in the planning domain and the users were asked to create the mappings and code generation for a similar object and operator in the planning domain.

The third stage of the study is used to evaluate Bowyer's ability to allow the user to specify a planning problem, get a plan from the planner and execute the plan in the virtual environment. A partially specified planning problem was provided and the users were asked

to complete the planning problem given a textual description of the literals to add to the initial and goal states. The users were then given general instructions on setting up Bowyer to use the planner to retrieve a plan and execute the returned plan in the virtual environment. The virtual environment was loaded for the user and running in the background through out the study.

Since the study was aimed at planning researchers and not virtual environment developers and to shorten the length of the study the users were not asked to create a Zócalo game client base class specification or enter methods or properties into the code library. The users were also provided with the Set State and Manage Plan document used by Zócalo and were not asked to create any other documents using the *Document Builder* interface.

5.3 Study Findings

Participant evaluation for the study was collected in two forms, a set of survey questions given at the end of the study and a set of open ended questions given through out the study at the end of each stage and the end of the exercise.

The survey questions were broken into sections based on demographic information, experience using Bowyer, Bowyer's functionality and Bowyer's applicability as a research tool. The demographic information questions were used to rate the user's experience in planning theory and application, planning tools and virtual environment use and code development. The experience using Bowyer questions were designed to evaluate the user's overall experience using Bowyer in term of user satisfaction. The Bowyer functionality questions were used to get the user's rating of Bowyer's ability to complete the tasks it was designed to accomplish. Finally the Bowyer applicability as a research tool questions were

designed to rate the user's likelihood to use Bowyer in their own research and to recommend Bowyer to their colleagues.

The open ended questions consisted of three questions asked at the end of each stage focused on only that stage's functionality and four questions at the end of the study used to give an overall evaluation of Bowyer. The three questions asked at the end of each section asked the participant to comment on the positive aspects of Bowyer functionality, the negative aspects of Bowyer functionality and what the participant would change or add to the provided Bowyer functionality for that stage. The four questions asked at the end of the study were aimed at getting the participant's evaluation of Bowyer's ability to accomplish the goals it was designed to accomplish. The participant was asked: if Bowyer would be a useful tool for integrating their planning research into virtual environments, if Bowyer was accessible to users that were familiar with planning techniques but not familiar with the development of virtual environments, what additional functionality they would like to have implemented in Bowyer and finally they were asked for any additional comments that did not fit into the other sections regarding their experience with Bowyer.

The studies findings are reported below. The results of the survey and open ended questions are broken into their corresponding stages. The results are reported alone in this section with discussion of the results in the next section.

Five participants took part in the study and demographic information was collected to determine how representative the study participants were of the target users. The participants were chosen to represent the range of target users as much as possible. The participants experience was largely skewed towards being very familiar with planning techniques and

application and less familiar with virtual environment development which represents the target user base for which Bowyer was designed. The participants were asked to rate their experience on a scale from 1 to 5 with 1 being no experience and 5 being an expert.

Table 5.1 Survey Results: Study Participants Demographic Information

Demographic Information	Average
Rate your experience with Artificial intelligence Planning (Theory)	4.4
Rate your experience with Artificial intelligence Planning (Application)	4.2
Rate your experience with Planning Tools	4.0
Rate your experience with Virtual Environments (Using)	3.4
Rate your experience with Virtual Environments (Developing)	2.4

The next section of survey questions covered the participants experience using Bowyer including their satisfaction with the functionality that Bowyer provides. The questions break the use of Bowyer into sections as well as assess the overall experience using Bowyer. The participants were asked to rate their experience using Bowyer on a scale from 1 to 5 with 1 being not satisfied or annoyed to 5 being very satisfied with the user experience.

Table 5.2 Survey Results: Bowyer Use

Bowyer Use	Average
Rate your experience to define planning domains	4.4
Rate your experience creating virtual environment code representations	3.8
Rate your experience creating a planning problem and retrieving a plan	4.4
Rate your experience executing the plan in the virtual environment	4.0
Rate your overall experience integrating planning into a virtual environment	4.4

The next section of the survey questions was used to evaluate the ability of Bowyer to complete the tasks for which it was designed. These goals include integrating planning research into virtual environment, aid the user in testing there planning research in the virtual

environment, aid the user in using the Zócalo framework, aid the user in testing planners and plans using the Zócalo framework and to be as accessible as possible to users that are familiar with planning research but unfamiliar with developing source code for virtual environments. The participants were asked to rate their evaluation of Bowyer's functionality on a scale from 1 to 5 with 1 being unsatisfied with the functionality provided by Bowyer and 5 being very satisfied with the functionality provided by Bowyer.

Table 5.3 Survey Results: Bowyer Functionality

Bowyer Functionality	Average
Rate Bowyer's ability to aid you in integrating planning research into virtual environment	4.6
Rate Bowyer's ability to aid you in testing planning research using virtual environment	4.6
Rate Bowyer's ability to aid you in using the Zócalo Planning Framework	4.8
Rate Bowyer's ability to aid you in testing planners and planning research using the Zócalo Planning Framework	4.6
Rate Bowyer's accessibility to non-technical users (planning researchers unfamiliar with developing code for virtual environments)	3.4

The last section of the survey questions was used to assess Bowyer's applicability as a research tool, including if the participants would use Bowyer in their own planning research and would recommend Bowyer to their colleges as well as how well the study gave the participants a general understanding of the functionality Bowyer provides and the tasks it can be used to complete. The numbers listed in the table represent the number of participants that selected that answer.

Table 5.4 Survey Results: Bowyer as a Research Tool

Bowyer as a research tool	Not Applicable	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Bowyer allows me to integrate planning research techniques into virtual environments					2	3
I would use Bowyer to aid me in my planning research				1	1	3
I would prefer to use Bowyer over the current techniques I use to accomplish these tasks					3	2
I would recommend Bowyer to my colleagues					2	3
I believe this study gave me a good general understanding of Bowyer's functionality and scope of use					1	4

A set of open-ended questions were also used to allow the participants to evaluate Bowyer and provide their own feedback. The questions were broken into four sections to correspond to the three stages of the study and a set of questions to evaluate Bowyer as a whole. The questions for the three sections each asked the participants to access the positive aspects of Bowyer functionality, the negative aspects of Bowyer functionality and what aspects of Bowyer functionality they would change or improve.

The overview questions asked the participants to assess how well Bowyer achieved the goals for which it was designed: integrating planning research into virtual environments and making this process accessible to users that were familiar with planning research but unfamiliar with development of source code for virtual environments. The participants were also asked for any overall changes, improvements or additions they would like to see to Bowyer as well as any other additional comments that did not fit in the other sections. A

summary of the participant's responses to the open-ended questions, focused on common responses, is presented below.

Stage 1

As covered in the study setup subsection, stage 1 of the study was used to evaluate Bowyer's ability to allow the participants to define a planning domain.

The positive aspects of Bowyer's functionality listed by the study participants were that it displayed the information in a useful format so that all information that was needed when working with Bowyer was available. Participants also believed that the way Bowyer allowed them to specify a planning domain was intuitive and matched well with the STRIPS planning representation it is based on.

The negative aspects of Bowyer's functionality listed by the study participants were mainly centered on the fact that the tabbed interface created hard lines between the definition of types, literal, objects and operators and that the flow between them may not be logical to some users. They believed most users would want to create operators first and define types, literals and objects as they needed them to define the domain. The participants also listed some usability concerns such as realizing the parent-child relationship in the type editor and that the interface was not efficient for mass data loading.

The aspects of Bowyer functionality that the participants would change or add were all usability concerns such as changing the interface so there are not hard lines drawn between the creation of planning domain attributes and including more tool tips to guide users.

Stage 2

As covered in the study setup subsection, stage 2 of the study was used to evaluate Bowyer's ability to allow the participants to create virtual environment representations of the planning domain's operators and objects.

The positive aspects of Bowyer's functionality listed by the study participants were that it helped give them a better understanding of how the virtual environment they were using worked, the mapping process was straight forward once they understood it through the help of the user manual and many believed that with more use the mapping interfaces would prove to be efficient.

The negative aspects of Bowyer's functionality listed by the study participants were that the mapping interface was not as intuitive as the other interfaces in Bowyer, the flow and order to complete steps was not always intuitive and they believed the mapping process required an understanding of the virtual environment slightly beyond that of a novice to virtual environment development.

The aspects of Bowyer's functionality that the participants would change or add to Stage 2 included more information available to the user about the virtual environment and a set of usability problems such as improving the mapping interface to be more intuitive and follow a more natural flow of steps and guiding the user more when they are expected to complete certain tasks.

Stage 3

As covered in the study setup subsection, stage 3 of the study was used to evaluate Bowyer's ability to allow the participants to create a planning problem, get a plan from the planner and execute the plan in the virtual environment.

The positive aspects of Bowyer's functionality listed by the study participants were that it provided a good way to test planners and plans in the virtual environment, gave them a better understanding of how the planning was used in the virtual environment and provided an intuitive interface to the process of building a planning problem, retrieving a plan and executing it in the virtual environment.

The negative aspects of Bowyer's functionality listed by the study participants were centered on usability concerns such as more information about the planner's process of searching for a plan and more information about the connection and planning setup options.

The aspects of Bowyer's functionality the participants would change or add were all usability concerns such as highlighting the current step being executed in the plan graph displayed to the user and more guidance on moving through the tabs in the interface.

Overview

The overview open ended questions were used to evaluate how well Bowyer is able to meet the goals it was designed for and what additional functionality the participants would like to see added to Bowyer in the future.

The participants overall response when asked if Bowyer allowed them to integrate planning research into virtual environments was that it did provide the functionality needed, would be useful for giving demos of planning research and was generally easy to use especially if provided with more virtual environment information.

When the participants were asked if Bowyer was accessible to users that were familiar with planning research but unfamiliar with virtual environment development the responses were mixed with some participants believing that it was initially accessible, some believed it was accessible with use of the user's manual provided to them and the remaining believing that it would be accessible if given more virtual environment information and more time to work with Bowyer.

When asked for any changes or additions to the existing functionality and for any additional comments on Bowyer the participants mainly reiterated the usability concerns they listed in the previous stages with more virtual environment information provided, changing the tabbed interface, and providing more guidance to the user.

5.4 Discussion

This subsection discusses the possible reasons for the study findings and what future work is planned for Bowyer and its descendants as a result to address the study findings.

From the responses to the survey and opened ended questions, overall it is apparent that for target users Bowyer is a useful tool that is useful for integrating planning research into virtual environments. In general the participants were satisfied using Bowyer. They believe Bowyer provides useful functionality for defining planning domains and that it corresponds well with the STRIPS representation it is based on. They believe it provides useful functionality for creating virtual environment representations of the planning domain operators and objects and with more use of Bowyer would prove to be an efficient interface. Finally they believe Bowyer provides useful functionality to work with and test planner implementations as well as test plans in a virtual environment which would be useful for

creating demos of their planning research and future application of planning research to non-research based tasks.

There were some problems that will need to be addressed in the future development of Bowyer. These problems and possible solutions are addressed in the remainder of this section.

The study participants were told that the implementation of Bowyer they would be using was a proof of concept "alpha" release and to try to focus their responses more on the functionality provided and not the usability of the implementation. Despite this information, the majority of the negative aspects and the aspects of Bowyer that the participants would change or add were mostly concerning usability issues which show how important usability is to user's satisfaction. Given that Bowyer is in a proof of concept stage and a research tool, currently the provided functionality is held as a higher priority then its usability. Usability concerns will need to be addressed once the needed functionality has been determined and in the form of usability studies. I had this in mind when creating the initial implementation of the Bowyer tool and Bowyer is designed in the Model, View, Controller (MVC) design pattern [3][21] so that in the future the user interface, corresponding to the view, could be changed to address usability problems with as little change to the overall functionality and knowledge representation as possible.

Another fact to be taken into account is that all study participants had not used Bowyer before the study although some users had used Bowman which provides similar user interface and functionality to some parts of Bowyer. Bowyer is also a research tool and works with a complicated domain so some learning curve is going to be encountered

regardless of the usability of the tool. Additional usability improvements will continue to be made as they are encountered but given the context in which Bowyer will be used the goal of robust functionality and efficiency for experienced users will remain a higher priority the creating an intuitive interface for novice users.

This pilot study served the purpose it was designed for, giving initial target user evaluations of Bowyer to ensure the tool was generally on the right track and highlighting areas or improvement. One of the main changes planned in future work on Bowyer and its descendants is to change the tabbed interface currently used. More research will need to be done as well as usability studies to determine the best interface to use for these types of tasks to be more intuitive and to provide the user with more guidance while also being efficient for experienced users. This as well as smaller usability changes such as tool tips, layout of components, usage flow and other features to make the user interface more robust will continue to improve over time with more testing and user feedback.

Bowyer was designed with the main goal of aiding researchers in integrating planning research into virtual environments. From the study findings and the participants feedback it can be determined that Bowyer does a good job of accomplishing this goal. However Bowyer was also designed with a secondary, but related goal of being accessible to users with planning knowledge but little virtual environment development knowledge. It can be seen from the study findings and participant feedback that although the participants believed Bowyer is accessible to these users, more work is definitely needed in this area. Virtual environments are complex systems with large learning curves, Bowyer aims at helping flatten this learning curve to aid users in understanding and working with virtual

environments. Bowyer is able to do this using a combination of the functionality provided to work with virtual environments as well as providing documentation covering how specific virtual environments fit into Bowyer and the Zócalo framework. Bowyer's functionality and user interface will continue to improve with future development, but due to the large variation of virtual environments structure and design the largest improvement in this support will come from adding to the standard documentation provided with Bowyer and the Zócalo game clients for specific virtual environments. This documentation as well as the code library for each virtual environment will grow and improve over time with testing and feedback from users of Bowyer, its descendants and the Zócalo framework.

6 Discussion

This section provides a summary of the benefits provided to the user by Bowyer. Additionally, it provides a general analysis of how well Bowyer is able to address the two obstacles for which it was designed, integrating planning research into virtual environments by translating from the declarative planning domain to the procedural virtual environment domain and aiding non-developer planning resaerchers in incorporating planning into virtual environments. Finally, a discussion of future work and some concluding remarks are given.

6.1 Benefits Provided By Bowyer

Bowyer is designed to provide several benefits to the planning researcher and other related users as well as serve a variety of utility functions in the use of the Zócalo framework. As has been discussed throughout this thesis the main benefit Bowyer is designed to provide is to bridge the gap between the declarative planning domain and the procedural virtual environment domain. This provides the additional benefit of allowing planning researchers and game designers, with basic planning knowledge, to incorporate planning techniques and functionality into virtual environments to create more robust virtual environments, add dynamic narrative structure and allow new elements of narrative and game play to be integrated into the virtual environment more efficiently.

Along with the added benefits to virtual environments and games, Bowyer has been designed to aid planning researchers in testing planning algorithms, plans generated by planning algorithms and other aspects of planning such as interactive narrative in virtual environments. Bowyer is also designed to help the game designer to test the game's existing

code base and its support for the plans used in the Zócalo framework by determining if the game client contains sufficient operator and object representations that are generated by Bowyer or hand written by game developers. Additionally, Bowyer provides utility functions such as the document builder, planner integration and execution manager status monitor to help users setup and use the Zócalo framework more efficiently.

6.2 Evaluation of Bowyer

This subsection provides an evaluation of how well Bowyer is able to solve the problems for which it was designed. The evaluation is done under the understanding that Bowyer, at this point, is meant to serve as a proof of concept and not as a complete tool or solution to these problems.

6.2.1 Declarative and Procedural Domains

In general, Bowyer does achieve the goal of translating declarative planning domain specifications into procedural based representations of the planning domain for use in virtual environments. The purpose of this section is to evaluate how well Bowyer accomplishes this task and what can be improved.

Bowyer's ability to specify a declarative planning domain is based on STRIPS specification which is commonly used in planning research and has been proven as an efficient and sound way of specifying a planning domain; but it does have its limitations. Currently the functionality of Bowyer does not suffer from the limitations of this representation but as Bowyer's planning support is increased this choice of representation could reach its limitations. Future work is planned to extend Bowyer's planning specification functionality. Bowyer's planning domain specification functionality could also

benefit from planning domain verification techniques to verify that the domain being specified by the user is valid.

Bowyer's ability to specify the procedural virtual environment is more limited in terms of scope compared to its ability to specify planning domains. Bowyer is designed to work within the context of the Zócalo framework and accordingly only needs to be aware of the Zócalo game client for the virtual environment(s) being used. Currently this functionality has proved sufficient for Bowyer's interaction with virtual environments through Zócalo but future needs may prove more robust virtual environment representation are needed within Bowyer.

Bowyer's mechanism for bridging the gap between declarative and procedural domain is to use code generation to generate the procedural representation of the declarative planning operators and objects. A XSLT template with XML specification approach to code generation was chosen to allow support for additional virtual environments through the addition of templates, client specifications and code libraries for those environments. This gives Bowyer the ability to add support for additional virtual environments without the need to modify Bowyer. Although this approach accomplishes this goal it also moves the majority of the logic for the code generation process to the XSLT templates for the environment and limits the type of action classes that can be created by the information available in the static XML representation generated by Bowyer.

6.2.2 Aid for Non-Developer Users

Bowyer allows non-developer users, users familiar with planning research but unfamiliar with virtual environment development, to integrate planning research into virtual

environments. This is achieved by allowing them to create procedural representations of the planning operators and objects without requiring the knowledge of how to develop the source code for them for the environment. This ability is based on a few assumptions that need to be fulfilled. Non-developer users require that virtual environment developers have supplied them with the Zócalo game client, base class specification and a sufficient code library for the environment being used to cover their planning domain. The Zócalo game client and base class specification can be built at the same time and will be created when a virtual environment becomes supported by the Zócalo framework. These two components can then be downloaded and used with Bowyer to interact with that environment. The code library for an environment will grow over time and mature through use. This will be the bottleneck of planning support for Bowyer; dependent on the ability of the code library to contain sufficient code modules to represent the requirements of operators and objects in the planning domain. As initially created code libraries begin to grow and mature, code libraries for new virtual environments will become easier to specify by using the existing code libraries for other virtual environments as a guide to the type of generic code modules needed by the planning process.

Bowyer's ability to aid non-developer users is also based on the assumption that all of the operators and object representations used in the environment will fit into the template patterns that correspond to the STRIPS template of checking preconditions and constraints, executing the operator to cause the desired effects and checking the effects to determine if the execution was successful. For the majority of classical planning support this is a reasonable assumption.

Although Bowyer is designed to allow users with limited technical knowledge and limited virtual environment knowledge to integrate planning techniques into the virtual environment there is knowledge specific to each environment that will have to be provided to the users so they can understand common practices for that virtual environment. This is a result of the unique representations most virtual environments choose to use to create their virtual worlds. It is assumed that this information is included in documentation that is included as part of the documentation of the Zócalo game client, base class specification and code library available for that virtual environment. In addition to this, information about the objects that exist in each virtual world, level or map, used in the virtual environment must be provided to the user, which is expected to be created when the map is created. There is future work planned to create standard templates for this documentation.

6.3 Future Work

Building systems and tools to integrate artificial intelligence planning into virtual worlds requires solving many problems associated with virtual simulations of environments, creation of narrative and user interactions in that narrative and the various approaches different researchers take to planning research. The main problems faced by systems and tools aimed at integrating planning into virtual worlds include: uncertainty, dynamic environments, distributed plans and teamwork, invalidation of plans during execution, maintaining environment data, authorability and constraints on plan content [4]. Bowyer's initial aim is based on solving some of the problems associated with authorability. With Bowyer being a part of the larger Zócalo framework many of the additional problems are

being addressed with initial attempts at solutions to these problems being implemented in Zócalo.

The work done to this point in the implementation of Bowyer is met to serve as a proof of concept for a solution to the problems of authorability it was designed to address. Given this, there are many areas of future work possible for Bowyer and tools of this nature. The future work can be divided into the separate stages used by Bowyer to bridge the gap between declarative and procedural domains.

The approach Bowyer takes to the specification of planning domains can be further improved by adding validation checks, adding decomposition support for hierarchical plans and adding mediation support to the planning domain specification. Validation support would allow the user to verify that the specified domain created with Bowyer's interface is a valid planning domain. Decomposition support would allow Bowyer to take advantage of the hierarchical planning support available in the Zócalo framework. Finally the incorporation of mediation support into domain specification and Bowyer as a whole would help Bowyer and the Zócalo framework address the problems associated with uncertainty, dynamic environments and invalidation of plans during execution. Integration with the Bowman tool for interactive narrative would also improve the planning support for Bowyer and unify the two tools for use in the Zócalo framework.

The specification of virtual environment representations as Zócalo game client's can be improved by allowing more the one level of inheritance of the base classes in the client as well as allowing the user to use previously generated Bowyer classes to be the parent class of newly generated Bowyer classes. The code generation can be improved in several facets by

adding more intelligence to the code generation process. Currently one of the limiting factors of the code generation as discussed above is the support for multiple language generation through the use of XSLT templates for each client and building code libraries of methods and properties.

As well as improving the current virtual environment code generation an additional improvement to Bowyer's code generation is to allow Bowyer to generate planning languages such as PDDL in the same manner in which virtual environment source code is generated. This functionality and the ability to read other planning languages would add additional means of utilizing existing planning data in Bowyer as well as aid in Bowyer's adoption rate by planning researchers that use these other representations.

The plan generation and execution support can be improved by adding planning mediation support and making the plan execution support more robust. As mentioned before the plan execution support can be improved by adding mediation support to plan execution. This will allow Bowyer, through the use of the Zócalo framework, to further address the problems associated with uncertainty, dynamic environments and invalidation of plans during execution. Additionally, more plan execution functionality is possible for the Bowyer interface such as allowing the user to step backwards in the plan by allowing the user to "undo" an operator and as well as adding additional operators into the plan while it is being executed.

The addition of all of the functionality mentioned above is planned as future work in the development of Bowyer and its descendent tools.

6.4 Conclusion

Bowyer is meant to be a first step towards bridging declarative and procedural domains. It is designed specifically for planning domains and developed as a proof of concept implementation. Given this, Bowyer is able accomplish the two goals it was designed to achieve. It is able to create translations between declarative planning domains and procedural virtual environments and under the assumption that appropriate virtual environment client specifications and code libraries exist it is able to allow non-developer planning researchers to complete this translation process. Bowyer also serves several utility functions important to planning researchers and game designers for integration into the Zócalo framework. Bowyer and its descendent tools should prove to be useful to the development and application of planning research and as a part of the larger Zócalo framework will serve as a robust platform for integrating planning into virtual environments for research based as well as commercial projects.

7. References

[1] Bichler: L. A flexible code generator for MOF-based modeling languages, OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture

[2] Breen, M. Hendriks, M. Core UnrealScript Objects http://udn.epicgames.com/Two/CoreUnrealScriptObjects.html

[3] Burbeck, Steve Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC) http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

[4] Dini, D., van Lent, M., Carpenter, P., Iyer, K. Building Robust Planning and Execution Systems for Virtual Worlds. In Proceedings of Artificial Intelligence and Interactive Digital Entertainment (Marina del Rey, CA June 20-23rd 2006).

[5] Edelkamp, S., Hoffmann, J., PDDL2.2: The language for the classical part of the 4th international planning competition. Tech. Rep. 195, 2004 Albert-Ludwigs-Universit¨at, Institut f¨ur Informatik, Freiburg, Germany

[6] Fikes, R. E. Nilsson, N, J. STRIPS: A new Approach to the Application of Theorem Proving to Problem Solving. Artificial Intelligence 2 (1971) 189-208

[7] Fox, M., Long, D., 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. Journal of Artificial Intelligence Research 20, 61–124.

[8] Fox. M Long, D. The Automatic Inference of State Invariants in TIM. Journal of AI Research. vol. 9. 1998. pp. 367-421

[9] Ghallab, M. Howe, et al. “PDDL -- The Planning Domain Definition Language.” In Proc. of AIPS'98, 1998.

[10] Harris, Justin. Proactive mediation in plan-based narrative environments. Masters' Thesis, NC State University, 2005. http://liquidnarrative.csc.ncsu.edu/pubs/harris-thesis.pdf

[11] Hendler J, Tate A, Drummond M., AI planning: systems and techniques, AI Magazine, v.11 n.2, p.61-77, Summer 1990

[12] Herrington, J. Code Generation in Action. © 2003 Manning Publishing Co. 209 Bruce Park Ave. Greenwich, CT 06830

[13] Holman, K. G. Definitive XSLT and XPath. The Charles F. Goldfarb Definitive XML Series. © 2002 Crane Softwrights Ltds. Prentice-Hall Inc. Upper Saddle River, NJ 07458

[14] Howey, R. Long, D and Fox, M VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning using PDDL Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on Volume , Issue , 15-17 Nov. 2004 Page(s): 294 – 301

[15] Johnson, D., & Wiles, J. Computer Games With Intelligence. Australian Journal of Intelligent Information Processing Systems, 7, 61-68

[16] Laird, J.E. and van Lent, M. (2000). Human-level AI's Killer Application: Interactive Computer Games. AAAI Press

[17] Miliano, V. Moise, A Fall 2001 Unreal Engine Seminar Game and AI Handout http://udn.epicgames.com/Two/GameAndAIHandout.html

[18] Orkin, J. Agent Architecture Considerations for Real-Time Planning in Games. Proceedings of the Artificial Intelligence and Interactive Digital Entertainment. AAAI Press.

[19] Perrin, S. Benoit, E. Foulloy, L. Automatic code generation based on generic description of intelligent instrument 2002 IEEE International Conference on Systems, Man and Cybernetics. Volume: 6

[20] Redford, James Michael A Visual Tool for Generative Scripting in Computer Role-Playing Games. http://www.cs.ualberta.ca/~jonathan/Grad/redford.pdf

[21] Reenskaug, Trygve, The Model-View-Controller (MVC) Its Past and Present http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf

[22] Simpson, R. M.; et. al. GIPO: An Integrated Graphical Tool to support Knowledge Engineering in AI Planning. In Proceedings of the 6th European Conference on Planning. 2000

[23] Simpson, R. M.; et. al. GIPO II: HTN Planning in a Tool-supported Knowledge Engineering Environment. Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003), June 9-13, 2003, Trento, Italy

[24] Sweeney, T. Initial Guide to Unreal Game Engine. http://unreal.epicgames.com/UnrealFeatures.htm

[25] Sweeney, T. Initial Guide to Networking in Unreal Game Engine. http://unreal.epicgames.com/Network.htm

[26] Sweeney, T. Prestenback, R. UnrealScript Language Reference. http://udn.epicgames.com/Two/UnrealScriptReference

[27] Tate A., Hendler J., Drummond M. A Review of AI Planning Techniques. Knowledge Engineering Review Volume 1, Number 2, pages 4-17, June 1985

[28] Thomas, J. and Young, R. M. 2006. Author in the Loop: Using Mixed-Initiative Planning to Improve Interactive Narrative. In Proceedings of the ICAPS-06 Workshop on AI Planning for Computer Games and Synthetic Characters, Cumbria, UK.

[29] Vernieri, Thomas A Web Services Approach to Generating and Using Plans in Configurable Execution Environments Masters Thesis. North Carolina State University http://liquidnarrative.csc.ncsu.edu/pubs/vernieri-thesis.pdf

[30] Vrakas D, and Vlahavas, I. ViTAPlan: A Visual Tool for Adaptive Planning, in Proceedings of the 9th Panhellenic Conference on Informatics, (Thessaloniki, Greece, 2003), pp. 167-177.

[31] Weld, D. An Introduction to Least Commitment Planning. AI Magazine 15, 4 (1994), 27-61

[32] Weld D., Recent advances in AI planning. AI Magazine, summer 1999.

[33] Young R.M. et al., An Architecture for Integrating Plan-based Behavior Generation with Interactive Game Environments. Journal of Game Development, vol. 1, 2004

[34] Young, R.M. Pollack, M.E and Moore, J.D. Decomposition and Causality in Partial-Order Planning, Proc. 2nd Int'l Conf. AI Planning Systems (AIPS-94), AAAI Press, 1994, pp. 188-194.

[35] Young, R M. A Developer's Guide to the Longbow Discourse Planning System.. University of Pittsburgh Intelligent Systems Program Technical Report 94-4.

[36] Zou, Y. and Kontogiannis, K.. Towards a Portable XML-based Source Code Representation. In Proc. XML Technologies and Software Engineering (XSE2001) (2001), 343{353.

APPENDICES

8. Appendices

The sections of this chapter contain supplement information to aid in the understanding of Bowyer and the processes it uses for code generation.

8.1 Code Generation XML Schemas

The schemas below outline the XML schemas used for the planning object and operator specifications that are written out by Bowyer to be process by XSLT templates during the code generation process.

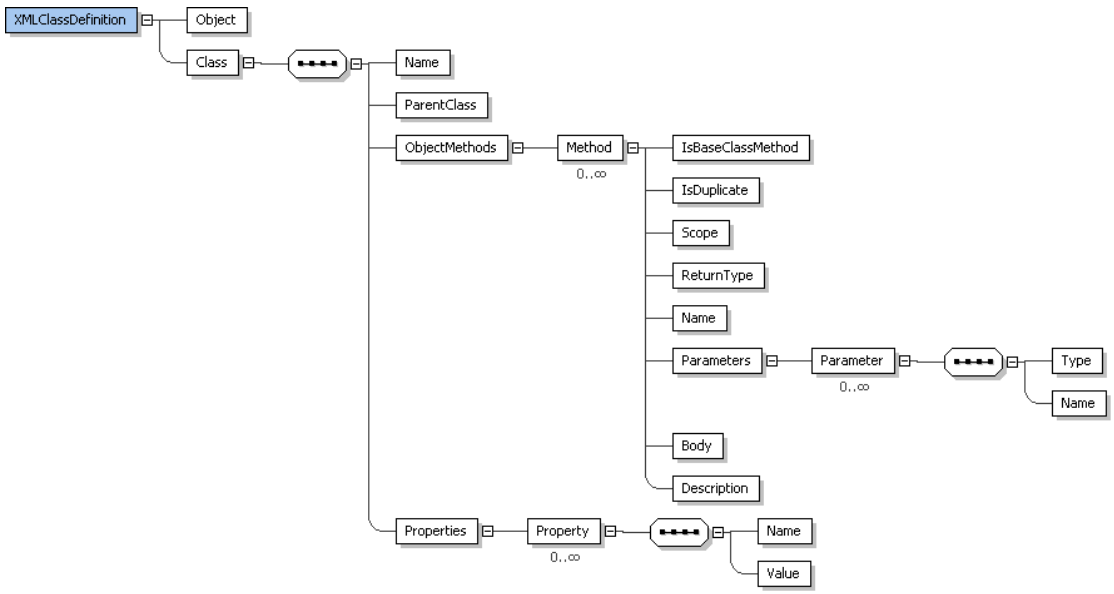

Figure 8.1 Object code generation XML schema

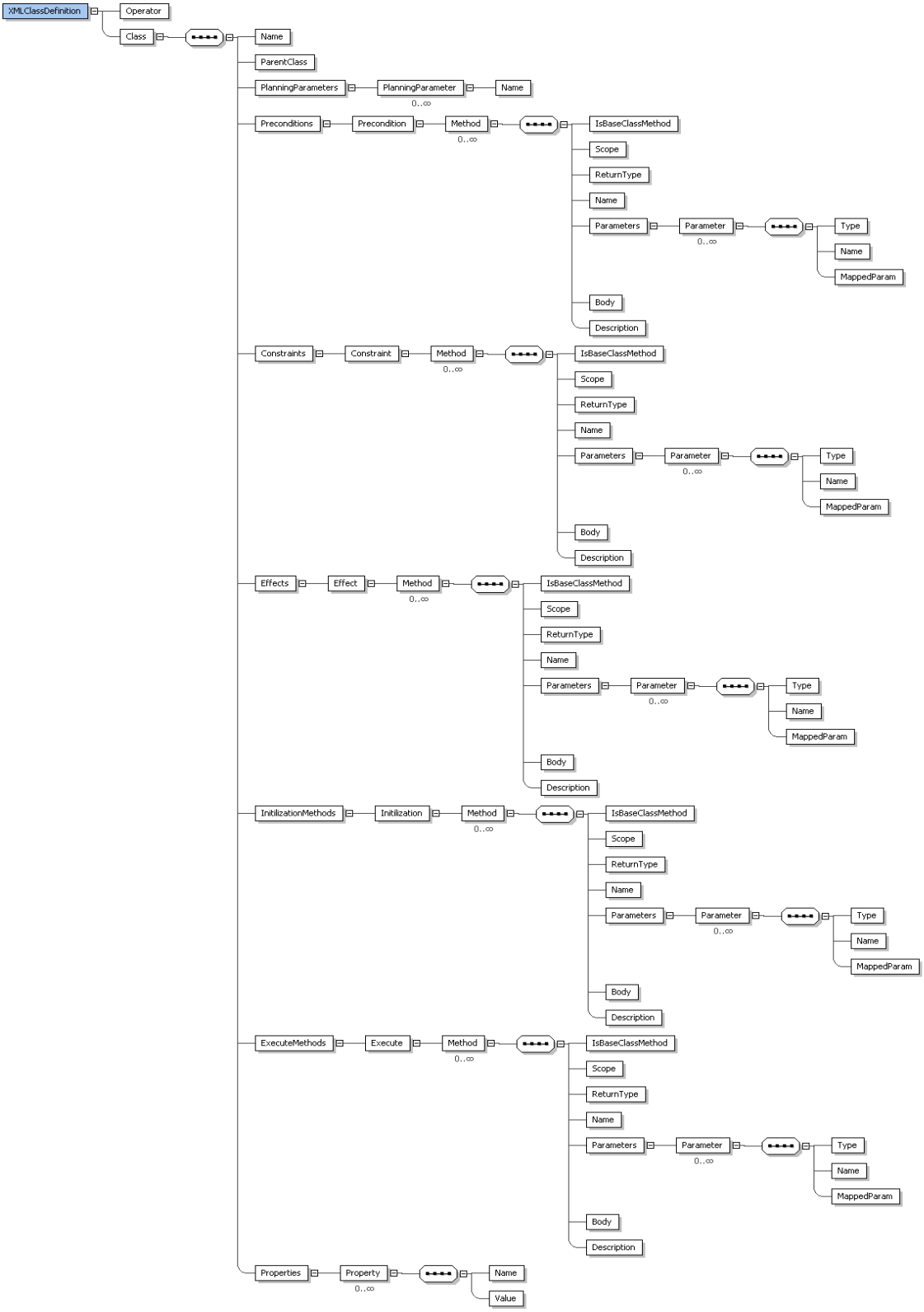
XMLClassDefinition
Operator
Class
Name
ParentClass
PlanningParameters
PlanningParameter
Name
0..∞
Preconditions
Precondition
Method
0..∞
IsBaseClassMethod
Scope
ReturnType
Name
Parameters
Parameter
0..∞
Type
Name
MappedParam
Body
Description
Constraints
Constraint
Method
0..∞
IsBaseClassMethod
Scope
ReturnType
Name
Parameters
Parameter
0..∞
Type
Name
MappedParam
Body
Description
Effects
Effect
Method
0..∞
IsBaseClassMethod
Scope
ReturnType
Name
Parameters
Parameter
0..∞
Type
Name
MappedParam
Body
Description
InitilizationMethods
Initilization
Method
0..∞
IsBaseClassMethod
Scope
ReturnType
Name
Parameters
Parameter
0..∞
Type
Name
MappedParam
Body
Description
ExecuteMethods
Execute
Method
0..∞
IsBaseClassMethod
Scope
ReturnType
Name
Parameters
Parameter
0..∞
Type
Name
MappedParam
Body
Description
Properties
Property
0..∞
Name
Value

Figure 8.2 Operator code generation XML schema

8.2 Example XML Generated by Bowyer

Example generated XML for the operator "ZAction_ZController_MoveToTarget" is presented below. The generated XML conforms to the operator schema presented above and is used by XSLT templates to generate the source code for the operator.

```
<XMLClassDefinition>
 <Operator>ZAction_ZController_MoveToTarget</Operator>
 <Class>
  <Name>MoveTo</Name>
  <ParentClass>ZAction_ZController</ParentClass>
  <PlanningParameters>
   <PlanningParameter>
    <Name>pawn</Name>
   </PlanningParameter>
   <PlanningParameter>
    <Name>target</Name>
   </PlanningParameter>
   <PlanningParameter>
    <Name>from</Name>
   </PlanningParameter>
  </PlanningParameters>
  <Preconditions>
   <Precondition>
    <Method>
     <IsBaseClassMethod>True</IsBaseClassMethod>
     <IsDuplicate>False</IsDuplicate>
     <Scope>final</Scope>
     <ReturnType>bool</ReturnType>
     <Name>IsTouching</Name>
     <Parameters>
      <Parameter>
       <Type>Actor</Type>
       <Name>Other</Name>
       <MappedParam>target</MappedParam>
      </Parameter>
     </Parameters>
     <Body />
    </Method>
   </Precondition>
```

```
    <Precondition>
      <Method>
        <IsBaseClassMethod>False</IsBaseClassMethod>
        <IsDuplicate>False</IsDuplicate>
        <Scope>function</Scope>
        <ReturnType>bool</ReturnType>
        <Name>IsZController</Name>
        <Parameters>
          <Parameter>
            <Type>Object</Type>
            <Name>ObjectToTest</Name>
            <MappedParam>pawn</MappedParam>
          </Parameter>
        </Parameters>
        <Body>if(ObjectToTest != null)
{
       if(ObjectToTest.IsA('ZAction_ZController'))
       {
            return true;
       }
}
return false;</Body>
        <Description>tests to determine is ObjectToTest is a ZController object</Description>
      </Method>
    </Precondition>
    <Precondition>
      <Method>
        <IsBaseClassMethod>False</IsBaseClassMethod>
        <IsDuplicate>False</IsDuplicate>
        <Scope>function</Scope>
        <ReturnType>bool</ReturnType>
        <Name>IsActor</Name>
        <Parameters>
          <Parameter>
            <Type>Object</Type>
            <Name>ObjectToTest</Name>
            <MappedParam>target</MappedParam>
          </Parameter>
        </Parameters>
        <Body>if(ObjectToTest != null)
{
    if(ObjectToTest.IsA('Actor'))
    {
```

```
        return true;
    }
}
return false;</Body>
        <Description>test to determine if the ObjectToTest is an actor object</Description>
      </Method>
    </Precondition>
    <Precondition>
      <Method>
        <IsBaseClassMethod>False</IsBaseClassMethod>
        <IsDuplicate>True</IsDuplicate>
        <Scope>function</Scope>
        <ReturnType>bool</ReturnType>
        <Name>IsActor</Name>
        <Parameters>
          <Parameter>
            <Type>Object</Type>
            <Name>ObjectToTest</Name>
            <MappedParam>from</MappedParam>
          </Parameter>
        </Parameters>
        <Body>if(ObjectToTest != null)
  {
      if(ObjectToTest.IsA('Actor'))
      {
          return true;
      }
  }
return false;</Body>
        <Description>test to determine if the ObjectToTest is an actor object</Description>
      </Method>
    </Precondition>
  </Preconditions>
  <Effects>
    <Effect>
      <Method>
        <IsBaseClassMethod>True</IsBaseClassMethod>
        <IsDuplicate>False</IsDuplicate>
        <Scope>final</Scope>
        <ReturnType>bool</ReturnType>
        <Name>IsTouching</Name>
        <Parameters>
          <Parameter>
```

```
          <Type>Actor</Type>
          <Name>Other</Name>
          <MappedParam>target</MappedParam>
        </Parameter>
      </Parameters>
      <Body />
    </Method>
  </Effect>
  <Effect>
    <Method>
      <IsBaseClassMethod>False</IsBaseClassMethod>
      <IsDuplicate>False</IsDuplicate>
      <Scope>function</Scope>
      <ReturnType>bool</ReturnType>
      <Name>!IsTouching</Name>
      <Parameters>
        <Parameter>
          <Type>Actor</Type>
          <Name>Other</Name>
          <MappedParam>from</MappedParam>
        </Parameter>
      </Parameters>
      <Body>local Actor temp;

  foreach RadiusActors(Other.Class, Temp, Agent.Pawn.CollisionRadius,
Agent.Pawn.Location)
  {
    if(Temp == Other)
      return false;
  }

  return true;</Body>
      <Description>Not version of IsTouching</Description>
    </Method>
  </Effect>
  </Effects>
  <ExecuteMethods>
    <Execute>
      <Method>
        <IsBaseClassMethod>False</IsBaseClassMethod>
        <IsDuplicate>False</IsDuplicate>
        <Scope>function</Scope>
        <ReturnType>void</ReturnType>
```

```
        <Name>MoveTo</Name>
        <Parameters>
          <Parameter>
            <Type>ZController</Type>
            <Name>Agent</Name>
            <MappedParam>pawn</MappedParam>
          </Parameter>
          <Parameter>
            <Type>Actor</Type>
            <Name>target</Name>
            <MappedParam>target</MappedParam>
          </Parameter>
        </Parameters>
        <Body>local vector LastSpot;
local vector StartSpot;

local Actor IntermediateTarget;
local Actor Waypoint;
                                StartSpot = Agent.Pawn.Location;
      LastSpot = StartSpot;

Agent.Pawn.SetMovementPhysics();
   IntermediateTarget = Agent.FindPathTo(target.Location);



      if(!bExecuted)
      {
         if (CheckPostConditions()) {
            MarkAsExecuted(true);
            return;
         }

         if (IntermediateTarget != None &amp;amp;&amp;amp; IsTouching(IntermediateTarget)) {
            IntermediateTarget = Agent.FindPathTo(target.Location);
            Agent.MoveTo(IntermediateTarget.Location, IntermediateTarget, true);
         }
         else {
            //I must be stuck
            if(IntermediateTarget != None &amp;amp;&amp;amp; Agent.Pawn.Location == LastSpot) {
               IntermediateTarget = Agent.FindPathTo(target.Location);
               Agent.MoveTo(IntermediateTarget.Location, IntermediateTarget, true);
            }
```

```
            LastSpot = Agent.Pawn.Location;
          }
        }


Agent.Pawn.SetPhysics(PHYS_None);
        Agent.Pawn.PlayWaiting();
        Agent.ExecutingAction = None;



    </Body>
        <Description>Method moves agent to the target</Description>
      </Method>
    </Execute>
   </ExecuteMethods>
   <Properties />
  </Class>
</XMLClassDefinition>
```

Figure 8.3 Example XML generated by Bowyer