

# ABSTRACT

VERNIERI, THOMAS MICHAEL. A Web Services Approach to Generating and Using Plans in Configurable Execution Environments. (Under the direction of R. Michael Young)

The computational scope of artificial intelligence in games has traditionally been limited by the processing requirements of the game engine's graphics and physics components. The emerging genre of interactive narrative typically relies upon AI planning systems that perform computation too demanding to integrate into commercial games. This thesis describes Zocalo, a collection of service-oriented applications, in which a planning Web service generates interactive storylines for story-based games and interactive narratives. The interfaces of the planning services allow for usage scenarios ranging from simple to complex. We describe the use of planning services both for run-time construction of narrative plans and for design-time iterative specification of game contents.

Zocalo facilitates the execution of plans in commercial game engines with only small modifications to the original games. It provides for the execution of story plans in numerous situations, automatically adjusting the state of the game's environment so that it is compatible with the beginning of the story. The plan execution functionality of Zocalo is intended for gaming environments but can also be applied to other applications in need of narratives.

**A WEB SERVICES APPROACH TO GENERATING AND USING PLANS IN  
CONFIGURABLE EXECUTION ENVIRONMENTS**

by  
**THOMAS MICHAEL VERNIERI**

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

2006

**APPROVED BY:**

---

Dr. Dennis Bahler

---

Dr. James Lester

---

Dr. R. Michael Young  
Chair of Advisory Committee

*To Penney, who has given me tremendous support and love throughout our  
relationship*

# BIOGRAPHY

Thomas Vernieri, who has always gone by the name Tommy, grew up in River Vale, New Jersey. When Tommy was nine years old, he moved to Wilmington, North Carolina with his family—parents Tom and Michele and his sister Elena. In high school, Tommy often made short films with his friends and considered a career in the film industry. He also enjoyed computer programming, which he first experienced in an academic setting as an elementary school student and again in high school under dual-enrollment at the University of North Carolina at Wilmington. Tommy graduated as the valedictorian at E.A. Laney High School in the year 2000.

After high school, Tommy moved to Raleigh, North Carolina and began work towards a bachelor's degree in Computer Science, with a minor in Film Studies at North Carolina State University. During his undergraduate work, Tommy joined the Liquid Narrative Group, a research group lead by Dr. Michael Young with a focus on the merger of artificial intelligence research and computer gaming. Tommy completed his BS degree in the fall semester of 2003, graduating *summa cum laude*. As an undergraduate, Tommy was admitted to the Accelerated Bachelor's/Master's program, and upon graduation he continued his work with the LN Group in pursuit of a master's degree. While working on his master's thesis, Tommy was a teaching assistant for classes in both Knowledge Discovery and Software Engineering, which he found to be an invaluable experience. Tommy completed the requirements for his MS in Computer Science during the 2006 spring semester.

Following the completion of his master's degree Tommy moved to Charleston, South Carolina. In Charleston, he will begin a career at Blackbaud, which specializes in software and services for nonprofit organizations. He is excited to begin work at Blackbaud, Inc. and

even more excited to begin a new stage of his life with Penney, his fiancée with whom he is engaged to be married in October.

# ACKNOWLEDGMENTS

This thesis would not have been possible without the help of many others. I thank David Christian, Mark Riedl, and Kevin Vaughn for supporting me when I joined the Liquid Narrative Group. While developing Zocalo, Justin Harris was an extraordinary help as we discussed many ideas, from grand designs down to mundane details. I also benefited from a steady flow of ideas, code, and friendship from Arnav Jhala, James Niehaus, and Jim Thomas. James Niehaus also helped increase the quality of this thesis by volunteering many small edits throughout. Matt Baker was a tremendous help in the final stages of actualizing my ideas through Unreal Tournament. I have enjoyed working with all of the members of the LN Group, and appreciate the work that many of them did to help build and test the Zocalo applications.

Thanks are also due to Dr. Dennis Bahler and Dr. James Lester for serving as my thesis committee members. Special thanks go to Dr. R. Michael Young, who served as my thesis committee chair and has been my advisor throughout my time as a graduate student. Dr. Young helped me redirect my research efforts numerous times, eventually leading to a topic that I could adopt wholeheartedly. I also appreciate that he fosters a community in the LN Group that is social and conducive to research, both at the same time. Outside of the LN Group, I was a teaching assistant for Dr. Laurie Williams for two semesters and I thank her for her understanding when my thesis won the battle for my time. She is responsible for much of my knowledge in the area of software engineering. The path from my early years at NC State University to my graduation was made easier by the always-helpful advice of Ms. Joyce Hatch and the solid foundation of object-oriented programming that I learned from Fred George, who taught my introductory programming.

Finally I would like to thank my friends and family, especially my mother, Michele, who suggested corrections for nearly every page of this document. My friends have always given me a great environment where I could relax and my family has supported me in every way I could ask; they have been a huge positive influence throughout my life.

This work has been supported by National Science Foundation CAREER award 0092586 and by Microsoft Research's University Grants Program.

# TABLE OF CONTENTS

LIST OF FIGURES .....	ix
LIST OF TABLES .....	x
1. INTRODUCTION .....	1
2. RELATED WORK .....	4
2.1. Plan-space planning .....	4
2.2. Planning Web services .....	6
2.3. The Mimesis architecture .....	8
2.4. Reactive mediation .....	10
2.5. Planning languages .....	14
3. OVERVIEW OF ZOCALO .....	16
3.1. Problems addressed by Zocalo .....	16
3.1.1. Resource needs of partial-order planning .....	16
3.1.2. Procedural execution of declarative plans .....	17
3.2. The design of Zocalo .....	18
3.2.1. Zocalo's central elements .....	18
3.2.2. The game engine's execution environment .....	20
3.2.3. The execution manager .....	21
3.2.4. The Fletcher Web service .....	23
3.2.5. The planning utilities assembly .....	23
4. IMPLEMENTATION .....	25
4.1. The execution manager implementation .....	25
4.1.1. Invocation and communication .....	26
4.1.2. Initialization .....	28
4.1.3. Action execution .....	30
4.1.4. Transition plans .....	33
4.1.5. Plan preparation .....	35
4.1.5.1. Common parameters .....	35
4.1.5.2. The manage plan message .....	39
4.1.5.3. The set state and manage plan message .....	45
4.1.6. Plan execution .....	46
4.1.7. Adding requested actions to an executing plan .....	48
4.1.8. Features for game development .....	52
4.2. An example execution environment implementation .....	53
4.2.1. Starting a plan-driven game .....	54
4.2.2. Executing actions .....	55

4.2.3. Detecting user actions .....	57
4.2.4. Proceeding after the end of the plan .....	58
4.3. The Fletcher Web service implementation .....	59
4.3.1. Creating a planning context .....	60
4.3.2. Retrieving planning context information .....	62
4.3.3. Exploring a search space.....	62
4.3.4. Using the database .....	63
4.3.5. Retrieving heuristic search function information .....	64
4.3.6. Implementing a Fletcher client .....	64
4.4. Implemented extensions to the Zocalo architecture .....	65
4.4.1. Mediation Web services.....	66
4.4.2. The Bowman planning client.....	66
5. DISCUSSION .....	68
5.1. Future work related to the Zocalo Web services .....	68
5.2. Future work related to plan execution .....	70
5.3. Conclusions.....	73
6. LIST OF REFERENCES.....	75
7. APPENDICES .....	78
7.1. Planning utilities schemas.....	78
7.1.1. Action specifiers .....	79
7.1.2. Domain definition .....	80
7.1.3. Environment state .....	82
7.1.4. Goal state .....	82
7.1.5. Planning context specification .....	82
7.1.6. Plan node definition .....	84
7.1.7. Plan space definition.....	86
7.1.8. Policy table.....	87
7.1.9. Predicates .....	88
7.1.10. Problem definition .....	89
7.2. Execution Manager – Socket Shell XML schemas .....	90
7.2.1. Received messages.....	90
7.2.2. Sent messages .....	92

## LIST OF FIGURES

Figure 3.1 The Zocalo architecture .....	19
Figure 3.2 The process of gameplay .....	22
Figure 4.1 Execution manager flow while processing a manage plan message .....	41
Figure 4.2 Execution manager flow while processing a new action request message during plan execution .....	51
Figure 7.1. The action specifiers document schema .....	79
Figure 7.2. The domain document schema, part 1 .....	80
Figure 7.3. The domain document schema, part 2 .....	81
Figure 7.4. The environment state document schema.....	82
Figure 7.5. The goal state document schema.....	82
Figure 7.6. The planning context document schema, part 1 .....	83
Figure 7.7. The planning context document schema, part 2 .....	84
Figure 7.8. The plan node document schema, part 1 .....	85
Figure 7.9. The plan node document schema, part 2 .....	86
Figure 7.10. The plan space document schema .....	87
Figure 7.11. The policy table document schema .....	88
Figure 7.12. The predicates schema.....	89
Figure 7.13. The problem document schema.....	90
Figure 7.14. Messages received by the EM-SS, part 1 .....	91
Figure 7.15. Messages received by the EM-SS, part 2 .....	92
Figure 7.16. Messages sent from the EM-SS.....	93

# LIST OF TABLES

Table 4.1 Action set interactions .....	39
---	----

# 1. INTRODUCTION

The breadth and depth of functionality that consumers are demanding from computer software grows everyday. This is especially true in the area of electronic gaming. As the graphics capabilities of computer game engines approach photo-realistic levels the use of artificial intelligence (AI) to enable new forms of game play will increasingly become a central distinguishing factor between titles. The demand for new types of interaction within game worlds will require the transfer of techniques used by academic AI researchers into the context of commercial computer game engines. When seeking to integrate a range of intelligent techniques into commercial computer game designs, game developers are faced with a dilemma: the computational requirements of typical AI algorithms place untenable demands on the resources available to users' machines running game engine code.

One new form of game play currently being explored by researchers in intelligent entertainment is that of *interactive narrative* in which a game's storyline is generated automatically and adapted in response to the user's actions at run-time. Many of the approaches to story generation use plan-based models where stories are composed by AI planning systems and translated into game-specific code for execution. The high computational cost of generative planning systems, however, places an unacceptable load on the CPU of a user's machine. This thesis describes *Zocalo*, a service-oriented architecture (SOA) for creating interactive narratives within existing commercial computer games in which plan generation is performed by Web services running on processors distinct from the game engine. *Zocalo* could certainly be used by a variety of applications in need of plans or plan execution, but the *Zocalo* architecture has been implemented and tested with interactive narrative gaming environments, which are the focus of this thesis.

Applications that directly use Zocalo's planning capabilities include in their code a lightweight client module responsible for communicating with *Fletcher*, Zocalo's planning Web service. Fletcher creates plans based on the client's instructions. Games use the output of Fletcher as a means to the end of executing a story plan, thus they can use the *execution manager*, an intermediary application that facilitates plan execution and communication with the Zocalo Web services. This service-oriented approach has a significant advantage over a design that requires a planning system be built into a game. Zocalo client games off-load the often-substantial computational cost of plan generation to a server rather than consuming computational resources on its own processor. In addition, applications other than the game engine (such as tools used by a game developer during game design) can connect to Fletcher or other planning-related services in Zocalo and interactively explore the planning data, exercising the full functionality provided by these Web services. Further, the distributed nature of Zocalo means that games and other applications that include a Zocalo client can discover planning services at runtime, allowing them to select a server with the lowest computational load, highest processor speed, or lowest usage rates.

Within Zocalo, the execution manager, which generally runs on the same machine as the game, allows a game to use Zocalo without directly invoking its Web services. The execution manager also handles much of the processing needed to execute a plan. Since this functionality is implemented in the execution manager, game developers that wish to execute plans can use Zocalo with only small changes to the game implementation. The open nature of Zocalo also gives game developers the option to directly use the Fletcher planning service, giving the developer substantial flexibility when it comes to controlling and gathering information about how the problem of generating a story is solved. When constructing a plan

in response to a request for a storyline, Fletcher uses an underlying model of planning as search through a space of plans. The messages that a client can send to Fletcher control how the search space of the planning problem is explored. During the development of a game, designers can interact directly with Fletcher servers to build and test game world specifications, requesting detailed information about the characteristics of the search spaces for given story contexts. Fletcher's accessibility allows game designers to configure their games precisely when determining trade-offs between numerous plan construction features.

Zocalo's provides different clients with several distinct methods for accessing its services and allows for the execution of plans in gaming environments. In the following chapter, we compare Zocalo to other related work. Chapter 3 details the need for Zocalo and gives an overview of the implementation. The details of the implementation are presented in Chapter 4. Finally, Chapter 5 discusses some limitations of the current implementation and possible future work; it is followed by a list of referenced publications and an appendix containing the data specifications used by Zocalo applications.

## 2. RELATED WORK

The implementation of Zocalo contains applications that build upon previous work in a variety of areas. The planner that underlies Fletcher implements a type of plan-space planning, summarized in Section 2.1. Like Fletcher, other planning implementations have exposed their interfaces via the Web, as discussed in Section 2.2. Shifting back to a broad view of Zocalo, Section 2.3 describes the Mimesis architecture, which inspired the Zocalo implementation. Reactive mediation, explained in Section 2.4, is a common element found in both Mimesis and Zocalo that allows these systems to guarantee that the goals of a story will be achieved despite a user's actions requests, which could interfere with the plan. Finally, we present approaches to representing data related to planning in a transmittable format since Zocalo applications and clients must share this type of information.

### 2.1. Plan-space planning

When computing plans for use in games or other applications, Fletcher uses a hierarchal causal link planner named *Crossbow*, a C# implementation of the Longbow planning system [22]; Crossbow uses *refinement search* [9] as a model of the planning process. Refinement search is a general characterization of the planning process as a search through a space of plans. A refinement planning algorithm represents the space of plans that it searches using a directed graph called the *plan space graph*; each node in the graph is a (possibly partial) plan. An arc from one node to the next indicates that the second node is a refinement of the first (that is, the plan associated with the second node is constructed by repairing some flaw present in the plan associated with the first node). In typical refinement search algorithms, the root node of the plan space is the empty plan containing just the initial

state description and the list of goals that together specify the planning problem. Nodes in the interior of the graph correspond to partial plans. Leaf nodes in the graph correspond to complete plans (solutions to the planning problem) or plans that cannot be further refined due, for instance, to inconsistencies within the plans that the algorithm cannot resolve.

Crossbow builds its search graph using best-first search. A heuristic search function ranks plans on the fringe of the graph and the most promising node is expanded next, its children plans placed on a queue of unexpanded nodes. The process repeats until a termination condition is reached. Typically, Crossbow planning tasks terminate as soon as a complete solution plan has been found or a size limit on the number of expanded nodes in the search space has been reached.

Crossbow’s planning algorithm combines partial-order, causal link representations (e.g., that of UCPOP [14]) with hierarchical planning approaches (e.g., [3, 18]). Details of the plan representation and the algorithm used to construct plans are beyond the scope of this thesis; specifics can be found in [22]. Several details, however, are significant for understanding a client application’s interaction with Fletcher. Crossbow (and thus Fletcher) represents all actions available in a given planning world as schematized operators; this set of operators is called the *domain*. A *plan* is a set of steps—that is, action instances instantiated from operator schemas—along with structures representing constraints over those steps. A *problem* is a tuple containing a complete specification of the current state of the planning world and a specification of the goals that a plan must satisfy in order to be considered *complete*.

A *planning context* is a triple consisting of a domain, a heuristic search function, and a plan space graph. Planning on Crossbow consists of

- 1) using the heuristic search function to rank all the plans associated with the nodes on the fringe of the plan space,
- 2) selecting the most promising node from the fringe, and
- 3) using the domain to create all of the plan's refinements and then adding those refinements to the plan space graph, creating a new planning context.

Planning is typically initiated by creating a planning context containing a graph with a single node whose plan has only two steps: an initial step encoding the current state of the problem and a terminal step encoding the goal state of the problem. Planning can also be initiated in Crossbow by seeding the plan associated with the initial node with additional steps and structures. This approach is useful, for instance, when a partial solution to a planning problem is already known.

## **2.2. Planning Web services**

Several other research projects have provided Web-based access to a planning system. A recent implementation of the O-Plan planner provides a Web-based interface accessible through CGI scripts [19]. At the time the interface was designed, no clear Web services standards had been generally agreed upon. Using CGI scripts increased the planner's accessibility, allowing clients to invoke it over the Web rather than as a local application. However, beyond the benefit of remote invocation, the use of CGI as a programmatic interface did not increase O-Plan's accessibility for remote client applications. The O-Plan interface is described solely through documentation that can be read by people, rather than in a format that can be processed by applications. Many Web services provide an interface description that can be processed by applications; this description is often expressed in the Web Services Description Language (WSDL), which provides a standard and structured way

of describing a remotely accessible application [2]. Developers of applications accessing services described in a WSDL document can use third-party tools to aid in the task of writing a client application; developers of O-Plan client applications must build their interface to the planner by hand based on specifications provided by user manuals that describe the CGI interface.

Tsoumakas et al. describe HAP-WS, a Web service that wraps around the HAP planning system [21]. In an approach similar to the one we use in the design of Fletcher and the other Zocalo Web services, HAP-WS provides a WSDL service description for the Web service, which receives and responds to messages using the Simple Object Access Protocol (SOAP) [1]. SOAP is a standard for messages containing typed data and its use is common among modern Web services. Since HAP-WS uses these two standards, developers can make use of existing developer tools to generate client code and client applications can easily discover the service at runtime.

Unlike Fletcher, however, HAP-WS is a relatively simple service that accepts only one message: a request for a solution to a planning problem. This request is composed of 1) a domain definition, 2) a problem definition, and 3) parameters for the plan discovery method used by the HAP planner. The service responds with text that describes the solution to the planning problem, which was computed by an instance of the HAP planner running on a server. While there are scenarios where this amount of functionality is sufficient for applications, there are a number of use cases where applications could benefit from a planning Web service with a wider range of features. Fletcher is such a Web service and the details of its functionality are described in Section 4.3.

Taking a broader approach, Hartanto and Hertzberg have developed a Web service that exposes the functionality of multiple planner implementations [7]. Their approach involves multiple Web services that work together allowing users to establish accounts and store domain information remotely. This Web service provides two alternatives that can be used to drive the planning process, which is more than HAP-WS but does not go to the same level of detail as Fletcher. The simplest way of using this service is for a client to send a small planning problem that also identifies the planner to be used; a solution to this planning problem is computed while the client waits, still connected to the service. The second option is used for planning problems that may take a long time to compute. The client first sends the planning problem and disconnects from the service immediately. The client periodically checks the service to find out if a solution has been computed; if a solution is ready, then the client retrieves it. For clients that will accept any solution to a plan, this set of services works well as a single framework for invoking a variety of planning algorithms. Due to this generality, the approach presented by Hartanto and Hertzberg does not provide the client with a way to control the details of the planning problem. Fletcher does provide this functionality, which is used by game developers when constructing game world initial states and domains.

### **2.3. The Mimesis architecture**

The concepts and applications in Zocalo build upon ideas that were first brought together in the Mimesis architecture developed by Young et al. [23]. The Mimesis architecture allows for the execution of plan-based behavior in a modified version of the Unreal Tournament commercial game engine. The three central parts of the Mimesis architecture are the Mimesis Controller (MC), the Mimesis Unreal Tournament Server

(MUTS), and the Mimesis Unreal Tournament Client (MUTC). The components in the MC have been reproduced as Zocalo Web services, with improvements in some cases. Most of the functionality found in the MUTS and the MUTC has not been precisely described in previous work; some of this functionality exists in the Zocalo execution manager with significant improvements. A detailed description of the Zocalo execution manager is found in Section 4.1.

The term *Mimesis Controller* identifies multiple intelligent components that run as separate processes, usually on separate machines. In the implementation of Mimesis there are three intelligent components, all of which use Longbow (mentioned in the previous section) as their underlying planner. The first component of the MC is the narrative planner, which generates a story plan; that is to say, a plan that contains steps that represent events in a story and correspond to actions that can be executed in the game. Next, the discourse planner augments the story plan with steps describing how the story should be presented. Finally, the speculative planner computes alternatives to the story plan for use in situations where a user's interactions could necessitate an adjustment to the original story. These components are invoked directly from the MUTS and their interfaces are not described in a language that can be processed by applications.

In Zocalo, these intelligent components have been replaced by new implementations that present machine-processable interface descriptions using WSDL. The narrative planner from Mimesis has been replaced by Fletcher, with added functionality that is used during game development (described in Section 4.3). A discourse planning application for Zocalo is under development by Jhala and Young [8]; presently, games using Zocalo are responsible for their own choice of a presentation method. All of the games that currently use Zocalo

present the story from a first-person perspective, through the eyes of the user-controlled character. *CrossWind* is a Zocalo Web service that performs reactive mediation, serving the same function as the Mimesis speculative planner; we describe reactive mediation in the following section. These Zocalo Web services are invoked from the execution manager rather than the game, allowing the complex functionality of plan management and execution (described in Section 4.1) to be reused in any game after only small modifications to the game implementation. In Mimesis, part of the MUTS is called the “Execution Manager object,” but it had only the most basic plan execution functionality. With the execution manager as a separate application in Zocalo, we are able to allow for the execution of plans in new types of scenarios and gain the added benefit of easy portability to new game engines in the future.

## **2.4. Reactive mediation**

During the execution of a story plan, if users are allowed to interact with the game, there is the possibility that a user’s actions will make further plan execution impossible. In both Mimesis and Zocalo, *reactive mediation* is used as the method of detecting such a situation and responding so that the goals of the plan can still be achieved while the user maintains a sense of agency in the game world. In Mimesis, reactive mediation was facilitated by the speculative planner developed by Riedl, Saretto, and Young [17]. Harris, under the direction of Young, implemented the same algorithm in the Zocalo Web service called *CrossWind* [5]. The process of reactive mediation allows users controlling characters in a game to feel as though they have an effect on the story, even in cases where their desired actions would interfere with the story plan. The following description of reactive mediation

goes into detail since it is an integral part of the execution manager, a core application in Zocalo.

Before a game can perform any action that does not come from the plan, it must be sure that executing the action will not make some later plan step impossible. For example, given a plan in which a game-controlled character will pickup an item, if the user shoots and kills that character before the pickup action has occurred then execution of the plan cannot continue after the point where the pickup step is finally reached. Before the execution of the plan begins, CrossWind detects the possibility of such situations by considering all of the actions that a user can request. Action requests such as these are said to be *exceptional* to the plan. If an action request is not exceptional, then either the action request is *constituent*, meaning that it is the same as a plan step scheduled to occur at the time of the request, or the action request is *consistent*, meaning that it is not a part of the plan yet it does not interfere with future plan steps.

Having determined what action requests are exceptional for each interval in the plan, CrossWind builds a *policy table* containing a list of responses for each exceptional action. A *response* is a course of action that can be taken in order to allow for the plan to complete execution while also taking the action request into consideration. The game could simply ignore any exceptional action requests, but doing so may leave the user disaffected, with a feeling that he or she is not participating in the story in a meaningful way. Reactive mediation responses are intended to avoid this feeling in the user by either allowing the requested exceptional action to execute with adjustments to the plan, called an *accommodation* response, or by allowing a substitute action to execute instead of the requested exceptional action, called an *intervention* response.

To compute an accommodation, CrossWind uses Fletcher to compute a new plan that can serve as a substitute for the original story. CrossWind instructs Fletcher to find an accommodation plan with the same goals as the original plan and with the exact same steps as those from the beginning of the original plan to the point where the exceptional action occurs. With these constraints, Fletcher attempts to find a plan that contains the exceptional step and can be substituted for the original plan. If it is able to find a plan, the plan is included as an accommodation response in the policy table. In some cases, Fletcher may be able to find many accommodation plans for an exceptional action, while in other cases it may find none before the plan interval passes.

To compute the intervention responses, CrossWind draws upon information in the domain of the planning problem (introduced in the section on partial-order planning, above). This domain contains an operator for each type of action that can occur in the game world; each operator may reference other operators that can serve as substitutes in an intervention response. These substitutes are called the operator's *failure modes*. A failure mode is generally an operator that a user may view as plausibly happening in the game world given the input that prompted the original action request. Which actions are plausible is up to the game developer to decide while constructing the domain. In our example of the user attempting to shoot another character, rather than a successful shoot action, the weapon could jam or the shot could miss its intended target. Both of these failure modes lack the effect of killing the target, which is what made the shoot action exceptional in our example. Each failure mode corresponding to the exceptional action is each listed as an intervention response in the policy table.

As mentioned earlier, execution of the plan cannot begin until the satisfaction of the plan's goals can be guaranteed; having the policy table nearly assures this for the original plan. If a policy table does not contain any usable responses for an exceptional action, then the game must do nothing in response to the user's action request in order to guarantee the satisfaction of the plan's goals. This is the situation that reactive mediation is designed to avoid, but lacking appropriate failure modes or enough time to find accommodations, there is no other known solution.

To make plan execution possible sooner, CrossWind computes policy tables iteratively. Since the planning process involved in computing accommodations may be time consuming, CrossWind first computes a policy table containing only interventions and then augments it with accommodations as Fletcher finds acceptable plans. When an accommodation response is used, the accommodating plan will have some steps that are different than those of the original plan. This means that responses for the original plan may not be valid for the accommodation plan and there may be situations in the accommodating plan where new exceptional actions could be requested. Just like with the original plan, before an accommodating plan can be executed its own policy table must be computed. To avoid a pause in execution during accommodation responses, CrossWind actually constructs a *mediation tree*, with a root node that contains the original plan and its policy table. For each accommodation in a policy table, CrossWind adds a child to that node in the mediation tree. The child contains the accommodating plan and its policy table. When an accommodation response is used, CrossWind makes the corresponding child node the new root of the mediation tree and prunes all of its siblings from the tree. The process of constructing the policy tree goes on throughout the plan's execution. The speculative planner in Mimesis was

named as such since one could say that it, like CrossWind, speculates about accommodations and interventions that might be needed if a user were to request a series of exceptional actions. CrossWind's relationship with the Zocalo execution manager is explained in Section 4.1.

## **2.5. Planning languages**

Zocalo applications and Zocalo clients transmit data relating to planning problems and solutions. Many other planning systems represent their inputs and outputs using PDDL [11] or similar languages and formalisms. Unlike these approaches, we use a custom data format to specify the data that is valid for communications with Zocalo applications. When Longbow was created, PDDL was not widely used; as a result, Longbow's inputs and outputs are expressed in its own Lisp-like data format. Crossbow uses the same algorithm as Longbow and thus uses similar inputs. The Crossbow application has not been updated to accept PDDL since the process of resolving subtle differences between the two representations of planning domains has not yet been generalized for use with any arbitrary domain.

In Zocalo, data is represented in XML, with semantics based upon the Lisp-like data formats used by Longbow. The Zocalo data formats are explained further in Section 3.2.5. Since Zocalo is based around the Web, the World Wide Web Consortium's XML is a natural choice for its data representations. Zocalo applications and Zocalo clients are able to manipulate and validate this data using tools that are built into many standard code libraries such as the System.Xml assembly in .NET, SAX in Java, and CL-XML in Lisp. The XML schemas that validate this data are referenced by the WSDL documents describing the Zocalo Web services. This means that clients can use the XML code libraries to validate the

messages they send to Zocalo Web services automatically; this is an advantage over systems like HAP-WS and the Web services developed by Hartanto and Hertzberg, in which data validity must be checked as a separate step.

It is possible that future work on PDDL will lead to an XML serialization or a higher-level ontology expressed in the Resource Description Framework (RDF), which is another World Wide Web Consortium standard [10]. McDermott and Dou have already addressed some of the technical hurdles that must be addressed in an RDF representation of planning domains [12]. If such ontology like this is created, it would facilitate translation between PDDL and the data formats used in Zocalo.

## **3. OVERVIEW OF ZOCALO**

The tasks of creating a planning problem and executing the resulting narrative plan in a gaming application are addressed by Zocalo. We first describe the complications that Zocalo addresses and then give an overview of Zocalo's design. The details of the applications involved in this design are presented in the following chapter.

### **3.1. Problems addressed by Zocalo**

Artificial intelligence planning has not been widely used in applications originating outside of the academic community. One reason for this is that integrating a declarative plan into an application that is predominately procedural is a complex task. Even after bridging this gap, the computational resources needed by a planner preclude its execution on many end-user machines, especially during the execution of a computationally expensive application like a computer game. These problems are addressed in the following sections.

#### **3.1.1. Resource needs of partial-order planning**

The process of constructing a plan by exploring a plan space requires a large amount of computational time and memory. As the planner refines partial plans, the set of plans it considers for future refinements grows, requiring more and more memory. The computational demands of planning prevent other computationally intensive applications from running on the same computer effectively. Improvements to heuristics and adjustments to the planning problem may reduce the resource needs to some degree. Even with these optimizations, common desktop computers lack the resources needed to effectively run a game and a planner at the same time. Attempting to do so is likely to result in perceptible slowdown of game's virtual world rendering, longer times spent searching for complete

plans, or both. If the planning problem is processed on a machine other than the one running the game, both applications benefit.

### **3.1.2. Procedural execution of declarative plans**

In most computer games, the world state is updated without deliberation, in response to user actions and other stimuli. We call this a reflexive mode of operation. For example, a user controlling a character that holds a ball can click a mouse button causing the character to throw the ball. Another type of reflex is initiated from within the game engine when the world state satisfies a particular set of conditions. As an example, consider a weak wooden box; when a heavy weight falls on the box it is crushed and destroyed. Some games also initiate actions using more complex logic. In a game where non-player characters (NPCs) are in competition with human-controlled characters, the game engine reflexively executes code for firing a weapon held by an NPC when the in-game logic dictates that it should happen. We say that games that operate in this way are executing actions procedurally. The game engine receives an instruction to execute an action and carries out the procedure for the execution of that action. In narrative games, the user understands a sequence of these actions as a story. Although an understandable story may emerge, the game executes individual actions independent of any knowledge about an overarching narrative.

While the games that we are considering handle each action in isolation, narrative planners, such as the one used in Zocalo, compute plans that declaratively state all of the actions that will occur from the story's beginning to its end. While some games developed specifically for AI research can use declarative plan representations directly [15], commercially available games are only designed to handle actions initiated in the ways described above. Building a game engine that understands plans is one approach to

combining AI research and computer gaming. Another approach is to use a commercially available game and initiate its built-in action execution reflex based on plan steps. This approach is what we use in Zocalo. The following section outlines the applications in Zocalo, describing how commercially available games can use Zocalo to drive plan execution.

## **3.2. The design of Zocalo**

We have implemented Zocalo to allow for the execution of partial-order plans in commercially available games while addressing the two complications described above. With Zocalo, plans are computed by servers running Web services. This leaves more resources available on the machine executing the game and also allows applications other than games to easily solve and explore planning problems. Zocalo also includes an application that, based on declarative plans, initiates procedural actions in commercial games, which have undergone only small modifications.

### **3.2.1. Zocalo's central elements**

To generate and execute novel storylines for games, Zocalo relies upon four central parts:

- An application library containing XML schemas and data utilities for the sharing of information among other Zocalo applications.
- Web services that compute story plans and perform additional plan-based analysis.
- An execution manager that initiates requests for story plans from the Web services, then uses the output of the services to manage the plans' execution within a game.

- An execution environment running within a game engine, responsible for translating action directives received from the execution manager into function calls specific to the game engine, running the functions used to implement plan steps and reporting back to the execution manager regarding the success or failure the functions' execution.

The overall Zocalo architecture is shown in Figure 3.1. Much of the game-side functionality found in Zocalo (e.g., the game-specific functions for action execution) was developed in previous work on the Mimesis system [23]. However, Zocalo's service-oriented nature makes our current approach more extendable and easier to integrate across disparate platforms and environments.

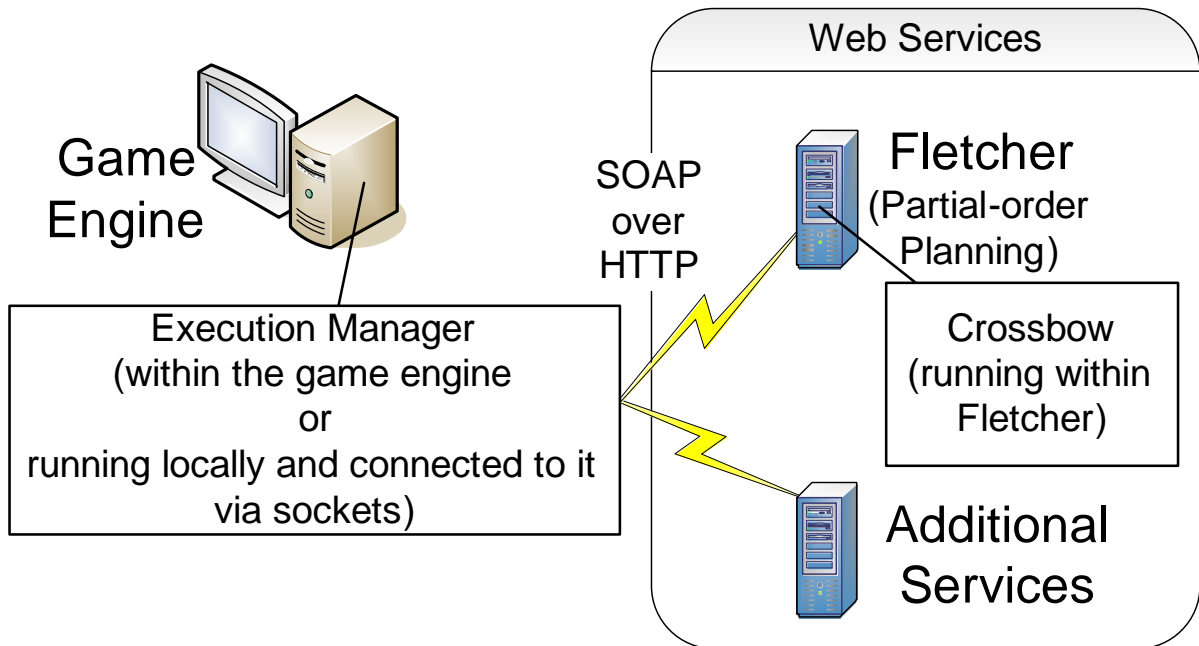


Figure 3.1 The Zocalo architecture.

### 3.2.2. The game engine's execution environment

Each game engine that makes use of Zocalo is extended with an implementation of a small interface. This interface allows the execution manager to instruct the game to execute an action. When it receives such an instruction, the game finds the functions responsible for performing that action in the game world and executes them. In addition to handling instructions to execute actions based on plan steps, the game must also request permission from the execution manager before performing any actions on its own initiative; ways in which actions could be initiated are described above, in Section 3.1.2, and can still apply in games that use Zocalo. In response to the game's request, the execution manager will either instruct it to execute an action or will inform it that its request was denied for one of the reasons explained in the following chapter. This denial notification is the second half of the interface that the game must implement. After the execution of an action halts, the game must inform the execution manager of its successful (or unsuccessful) completion. A game that uses Zocalo must be modified to support all of these interactions; collectively, these modifications are referred to as the *execution environment*.

We do not restrict the way a game implements the execution environment other than requiring that it implement the two-method interface. It can implement this interface as C/C++ functions, a .NET class, or a TCP endpoint that processes XML fragments. The game sends information, such as action requests, to the execution manager using the same technology that it chooses for exposing the execution environment interface. This choice of technologies makes it possible for many games to be modified to use Zocalo, even if the headers and source code used to compile the game are not available due to licensing restrictions. Since the interface is the only part of the execution environment that is restricted

by Zocalo, two games with different designs may use different execution environment implementations; the execution environment code could even be spread throughout many areas of the game's source code. As an example, Section 4.2, contains a description of the execution environment we implemented in Unreal Tournament 2004.

### **3.2.3. The execution manager**

Within Zocalo, the execution manager acts as an intermediary between the game engine's execution environment and the Web services needed to construct story plans. The typical usage of the execution manager has two phases, depicted in Figure 3.2 (more complex usage is possible and is fully described in the following chapter). First, as a game begins, the execution manager locates a Fletcher Web service and issues a request for a story plan. This request provides the planning service with a description of the game's current world state, the domain with actions available in the game, a specification of the preferences for the types of story plans it is requesting (in the form of the identification of a heuristic search function for the planning service) and a specification of the end or goal state of the story. Details on variations in the way the execution manager obtains plans begin in Section 4.1.4. Upon receiving a plan specification from the Web service, the execution manager transforms the step structure of the plan into a directed acyclic graph (DAG) with nodes representing the steps of the plan and edges representing the temporal constraints on the order of execution between those steps.

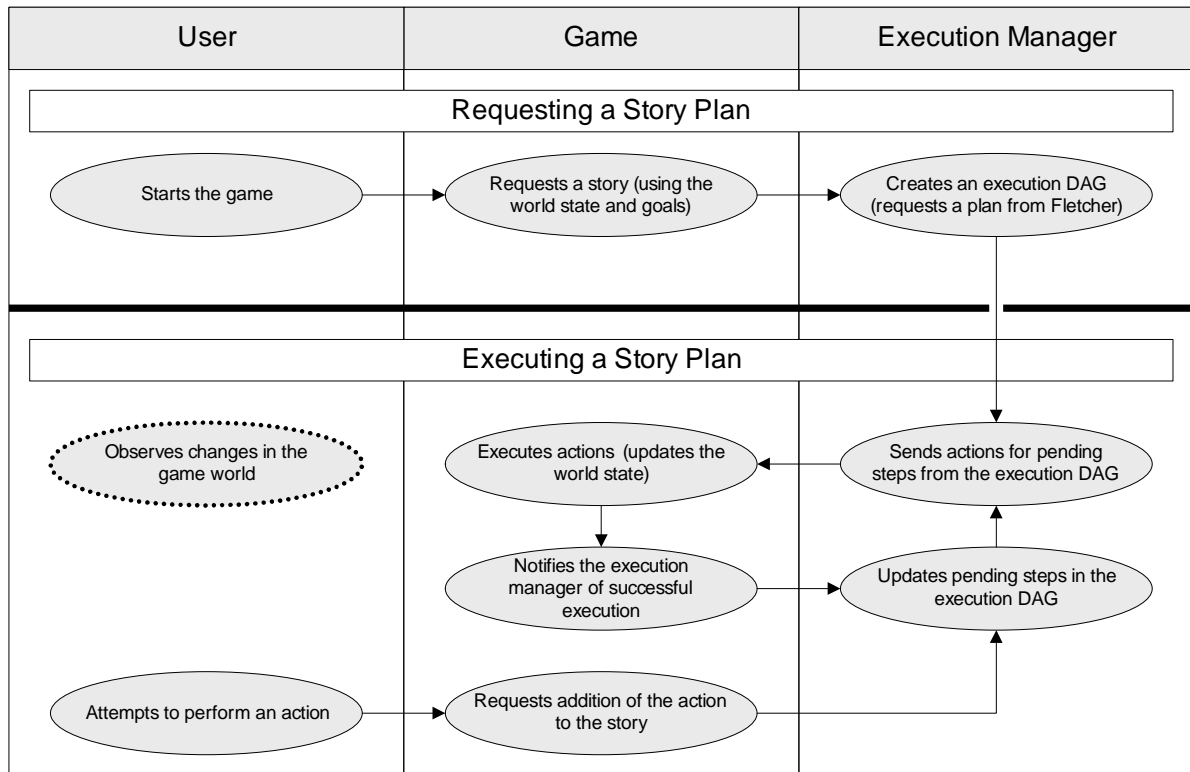


Figure 3.2 The process of gameplay. During the request for a story plan, control moves from left to right along the top of the diagram. The execution manager enters the plan execution phase as it begins sending steps to the execution manager. During plan execution, the game and the execution manager loop through the process of executing steps from the DAG; as a side effect (shown in a dotted oval), the user observes changes in the game world. At any point during plan execution, the user may attempt to perform an action, possibly resulting in the execution manager sending the action to the game for execution.

Once this execution DAG has been defined, the execution manager selects the minimal elements of the DAG—that is, just those actions that are not waiting upon other steps for their execution—and instructs the execution environment to execute those actions in the game world. Once the code for a step has executed, the execution environment notifies the execution manager of its completion and that step is removed from the DAG. This process repeats until all actions within the DAG have executed, at which point the story has reached its end.

### **3.2.4. The Fletcher Web service**

In order to obtain a story for the game, the execution manager first locates a Fletcher Web Service and issues a request for a story plan. At run time, a game's execution manager provides Fletcher with an operator library, a specific heuristic search function, and a description of the desired world state at the beginning and end of the story. Generating a plan in response to a specific story request is one of a number of capabilities that the Fletcher service exposes.

Planning developers working alongside game developers and story experts during the process of game development use Fletcher's additional functionality. These developers collaborate to construct the planning domains, heuristic search functions, and world state specifications that will be used as starting points and ending points for stories. Fletcher's service-oriented nature facilitates the development process by allowing game designers to connect to Fletcher and iterate through a process of domain specification and plan space exploration. The developer tool named *Bowman* facilitates this process and is described at the end of the following chapter.

### **3.2.5. The planning utilities assembly**

During the interactions between the Fletcher Web service and Zocalo clients, the shared data must be serialized into a format that can be transported across the Internet. All messages communicated between Fletcher and its clients are sent using SOAP over HTTP, with its inputs and outputs specified using XML documents defined in the XML Schema Definition Language [20]. These schemas are stored in a .NET assembly that we created, named *planning utilities*. The planning utilities class library is a part of Zocalo used by every other Zocalo application and some Zocalo clients. In addition to housing the definitive

version of Zocalo's XML schemas, it also provides the application developer with a range of classes that facilitate parsing and writing documents that conform to those schemas. The Zocalo schemas are too long to reproduce in this thesis but Appendix 7.1 contains visual depictions of many of the elements found in these schemas. Reuse of these schemas simplifies the task of creating new Web services such as CrossWind and Kyudo (introduced later, in Section 4.4.1), which have been added to Zocalo by other researchers. Planning utilities serves as a single location for the XML schemas used across Zocalo as well as providing an API for processing, validating, and manipulating planning documents.

## 4. IMPLEMENTATION

In this chapter, we describe the details of the core Zocalo application implementations. We begin by describing the implementation of the execution manager. The execution manager builds upon the functionality of Mimesis by adding transition plans; it can also be used by any game with only small modifications. The details of these modifications are outside of the scope of Zocalo, but we explain an example implementation to aid in a complete understanding of plan execution. These modifications are collectively called the execution environment implementation, and we describe the execution environment implemented in a commercial game, Unreal Tournament 2004. After discussing plan execution and the execution environment, we detail the interface exposed by Fletcher, which is used by the execution manager and game developers.

Finally, we present applications that have been added to Zocalo by other researchers. CrossWind, one of these applications, is used by the execution manager and mentioned throughout the chapter. Game developers use another one of these applications, called *Bowman*, during the development of games and planning problems. There is also a Web service called *Kyudo*, which implements an algorithm that builds upon the ideas found in CrossWind. All of these extensions are designed to coordinate with the core Zocalo applications, with the primary purpose of executing plans in gaming environments.

### 4.1. The execution manager implementation

The execution manager serves multiple functions within Zocalo, the most basic of which is providing execution environments with a simple way to use the Zocalo Web services and execute plans. Throughout this section, we describe messages sent from the

execution manager to the execution environment and messages sent from the game to the execution manager. The execution environment implementation is an inextricable part of any game that uses Zocalo, so the game and the execution environment are the same from the execution manager's perspective. The distinction is made because messages sent from the execution manager to the game go to one of the only two functions defined by the execution environment interface, but messages sent to the execution manager may originate from anywhere in the game's executing code.

The following details the execution manager's functionality, behavior, and use. We describe how the execution manager is invoked and initialized. After being initialized, it manages the execution of actions, which can originate from the game or from a plan. Finally, we describe features of the execution manager that are used during game development rather than during game play.

#### **4.1.1. Invocation and communication**

In order to use the execution manager's functionality, the game must either load it as a dynamically linked library (DLL) or invoke an external execution manager through TCP sockets. The execution manager is implemented in C#, which uses the .NET Framework; its interface is exposed as both a .NET DLL and a C/C++ DLL. In order to use these libraries, an existing game must be recompiled or linked with the execution manager DLL.

For games lacking publicly available source code and headers (such as Unreal Tournament 2004), we developed an application named the *Execution Manager – Socket Shell* (EM-SS) which sends and receives XML fragments via TCP sockets allowing the execution manager to be invoked from a game running as a separate process. The XML fragments used by the EM-SS are defined as the elements of the schemas in Appendix 7.2. In

addition to the functionality of the DLL interface, the EM-SS provides the game with a unique session identifier. If for any reason the connection between the game and the execution manager fails, it can be reestablished using this identifier. The EM-SS and the game may be run on the same machine or different machines on the same local area network, the latter scenario is not generally used since it would create an additional communications delay before any action could be executed. If the EM-SS is used by a game, the application must be started separately; this is easily done with a third application that, when executed, loads the game in the foreground and loads the EM-SS as a background process.

When the game invokes the execution manager through a DLL, all commands are communicated using function or method calls. When the EM-SS is used, the commands are expressed as XML fragments and sent through the TCP sockets connecting the game and the execution manager. Since the XML fragments are defined by a schema, the recipient is able to determine when a message has been completely received even if it has been broken into multiple packets by the TCP stream. In this section, we use the general term *message* to mean commands sent between the execution manager and the game regardless of the communications mechanism. These messages have various parameters for the needed data. Some of these parameters are integers, short strings, arrays of short strings, or true/false flags. All of the parameters used to pass complex data such as plans and domains use XML documents that conform to one of the planning utilities schemas. Whenever a complex parameter is first mentioned in this chapter it will be followed by a reference to the appendix subsection that contains a depiction of its validating schema. These same schemas are referenced in the description of Fletcher below.

### 4.1.2. Initialization

After connecting to the execution manager, the game must inform it of the current world state and provide it with the action schemas that will be used to alter the game world. The *set current state* message sent from the game to the execution manager includes an environment state document (see Appendix 7.1.3). This document is made up of a set of literals; a *literal* is a name, called the *predicate*, along with a sequence of zero or more object constants, called *terms*. For example, in a game world that contains a character called “robot” and a location called “underBridge,” a literal indicating the robot’s location may be expressed as “at(robot, underBridge).” This message from the game must fully describe everything that is true about the world state, and all other literals that can be formed to describe the world are assumed to be false (this is analogous to the closed world assumption commonly used in planning systems). It is also possible for the game to inform the execution manager of its current state and to request a plan as a single atomic action; the message responsible for this is later discussed in Section 4.1.5.3. Future games developed for use with Zocalo may be able to determine their own state at runtime; however, for all of the games that currently use Zocalo, this information is recorded by a game developer beforehand and then loaded by the game at runtime.

After setting the execution manager’s model of the current state, the game only modifies its state as instructed by the execution manager. This allows the execution manager to keep its model of the environment state consistent with the actual state of the game world. If the game does need to change its state without obtaining authorization from the execution manager, it may only do so when no plan is executing and there are no plans queued for execution. After changing its state, the game must inform the execution manager of its new

world state. This occurs when the user restores a previously saved game or loads a different game world setting. If the game must change its state in this way, it can send the *abort all plans* message to the execution manager to stop an executing plan. The game can send the abort all plans message at any time, but this is the only scenario used by current execution environment implementations.

Using this model of the environment's current state, the execution manager records any changes to the game world and updates its model of the environment state accordingly. In this way, the execution manager always has a model of the game's world state. The game must provide the execution manager with information about the actions that can happen in the game world so that the execution manager can correctly update its model of the environment state as they occur. This information is passed from the game to the execution manager using a domain document (see Appendix 7.1.2). A domain is made up of *operators*, which are schematized versions of actions that can be executed in the game world. Each operator is comprised of multiple sets of literals and possibly even more complex hierarchal structures which are used by Fletcher and other Zocalo Web services, but the execution manager only uses the sets of literals marked as the operators' effects. Whenever an action successfully executes in the game world, the execution manager determines which operator corresponds to the completed action and updates its model of the environment state with that operator's effects.

When a plan is being executed, the execution manager updates its model of the environment state using the operators defined in the domain of that plan. In addition, the execution manager can track the environment state between plans or without having ever executed a plan. To enable this feature, the game sends a domain document to the execution

manager with the *set outside-plan domain* message. When an action is executed, but no plan is being executed, the execution manager finds the corresponding operator in this outside-plan domain and updates its environment state model according to the operator's effects.

### 4.1.3. Action execution

After the execution manager is initialized, the game may request permission to perform an action. The request for permission is necessary during a plan's execution since occasionally a game that uses Zocalo may be restricted from executing some actions through the process of mediation (as described in Section 4.1.7). Outside of a plan's execution the request for permission is still necessary so that the execution manager can keep its model of the environment state consistent with the actual world state of the game. The response that the execution manager gives the game indicating that the requested action has been approved is the same type of response used to instruct the game that it should execute an action that originated as a plan step. Using only a single pattern of messages for executing actions regardless of their origination simplifies the implementation of the execution environment within the game.

The game's request for permission to execute an action takes the form of a *new action request* message containing the name of the operator corresponding to the action being requested (called the *operator name*), the names of the object constants that parameterize the operator (called the *bindings*), and an optional piece of data that may be used for the game's own needs (called the *request data*). If the execution environment needs to keep track of actions that it has requested or differentiate them from actions that originate as steps in a plan, the game may send any additional data in the request data parameter. The unaltered

request data will be sent back to the execution environment with the execution manager's response to the action request.

In response to the new action request message, the execution manager sends the execution environment either a *perform action* message or an *action refused* message. In the case of actions requested outside of a plan, the execution manager will always grant permission and will use the outside-plan domain to update the environment state model when the action succeeds. Usually an action requested during the execution of a plan is also permitted, in which case a perform action message would be sent to the execution environment; the process of determining the execution manager's response to an action request during plan execution is discussed in Section 4.1.5.1 and Section 4.1.7. The perform action message has the same operator name, bindings, and request data parameters as the new action request message, but also contains an additional integer parameter used as the *action identifier* and three true/false parameters, named *is initial step*, *is goal step*, and *is transition step*. The first two are used to identify the actions corresponding to the initial step and goal step of a plan, thus they are always false for perform action messages that come in response to action requests from the game. The *is transition action* parameter is also false for any perform action messages prompted by a request from the game; we explain transition plans in the next section. When the execution environment receives a perform action message at the beginning of a plan with the *is initial step* parameter set to *true*, it may perform tasks associated with the beginning of a plan's execution such as starting a scorekeeping mechanism or informing the player about the beginning of the story. Similarly, when the execution environment receives a perform action message with the *is goal step* parameter set

to *true*, it can perform tasks such as recording the player's score or asking the player what he or she would like to do next.

When the execution environment receives a perform action message with the *is initial* step and *is goal* step flags both set to *false*, it runs the game-specific code responsible for performing the action corresponding to the operator identified by the operator name parameter. After the action has completed its execution and its expected effects have been verified by the game, the execution environment sends the *action succeeded* message to the execution manager with the action identifier as its single parameter. If the game world's state is not consistent with the action's expected effects, then the *action failed* message is sent instead and also contains the action identifier as its only parameter. If an action fails to execute successfully, the execution manager has no way to recover since its model of the environment state may be inconsistent with the actual game world state. If this situation occurs, the execution manager clears its model of the environment's current state and is considered uninitialized until its model of the environment state is set by the game again.

When the game's action request comes during the execution of a plan, the execution manager may respond with an action refused message rather than a perform action message. This message contains as its parameters the unaltered request data as well one of a small set of error codes. The error codes are mentioned throughout this chapter and each corresponds to a particular reason why an action request would be refused. When an action request is refused, it is up to the game to decide what to do. If the action was the result of a user's input, simply not doing anything may frustrate the user as he or she will see no effect for the given input. The other extreme would be to inform the user that action cannot be executed; this too may frustrate a user who is not expecting the game to reveal the inner working of its

execution. Currently there is no generally satisfactory solution to this problem but some possibilities are discussed in Section 5.2.

#### **4.1.4. Transition plans**

At any point after the execution manager has been initialized with the current environment state, the game may instruct it to manage a plan. In order for a plan to begin execution, its initial state must exactly model the world state. In some cases this is trivial to achieve, but at other times the execution manager can be used to make this possible. The execution manager does this by executing two plans in sequence; first, the *transition plan* and then, the *main plan*. The transition plan is used to change the world state so that it is consistent with the desired initial state of the main plan. With this type of plan request, the game does not specify a full initial state for the main plan. Instead, it specifies a goal state for the transition plan. The world state that results from executing the transition plan is used as the main plan's initial state. Since the main plan is executed immediately after the transition plan, all of the literals that are in the transition plan goal state will be unchanged at the start of the main plan; the truth value of all other literals depends upon the state of the world at the beginning of the transition plan and the particular steps that are used in the transition plan.

When requesting a transition plan from Fletcher, the execution manager uses a separate set of operators, called the *transition domain*. At any point after invoking the execution manager, the game may provide it with these operators by sending them using the *set transition domain* message with a domain document. Using a separate domain allows the game to provide operators that are specialized for the task of quickly changing world state before a plan. The most common type of transition domain contains operators with corresponding actions that execute instantaneously; for example, during a main plan when a

game controlled character goes from one location to another a walk animation is played as the character gradually approaches the destination, but in a transition plan the character would just disappear from the starting location and appear at the destination. The choice of which operators to use in the transition domain is left up to the game rather than the execution manager, but the actual decision would most likely be made by the game developer during the game's design. If the game developer intends for the execution of the transition plan to be visible to the user, then operators used in the main plan domain can be reused for the transition plan domain. In addition to using operators defined in the transition domain, perform action messages that result from transition plan steps have their `is transition plan` flag set as *true*. This allows the game to know when the transition plan is executing and when the main plan begins. The process of executing these plans is described in Section 4.1.6.

The transition plan feature allows for scenarios that are not possible with other plan execution systems. In the previous work on the Mimesis architecture, the task of making the world state consistent with the initial state of the plan was always a responsibility of the game developer [23]. This resulted in either a limited range of possible planning problems for each game world or a large amount of work spent developing many detailed world configurations. With the Zocalo execution manager, the game developer may instead use the transition plan feature. The execution manager adjusts the world state at runtime rather than the game developer considering all possible initial states at design time, the game developer only needs to create a transition domain, which can be reused for any world configuration. Furthermore, this runtime capability makes it possible to execute more than one plan in series, even if the initial state of the upcoming plan is not the same as the state of the game world after the execution of the preceding plan. With the Mimesis system, there was no

guarantee that this would be possible since user actions that are consistent with earlier plans may leave the game in a state that is inconsistent with the upcoming plan's initial state.

#### **4.1.5. Plan preparation**

When the game instructs the execution manager to manage a plan, it uses one of two messages. If the game has changed its world state (e.g. by restoring a previously saved game) then it uses the *set state and manage plan* message, which, as an atomic operation, sets the execution manager's model of the environment state and begins the execution of a main plan. If the game has not changed its world state, then it may use the *manage plan* message, which results in the use of a transition plan when necessary before the execution of a main plan. The manage plan message may be used at any time after the execution manager has been initialized, even during the execution of another plan. If the message comes during another plan's execution, the execution manager does as much as it can to prepare the newly requested plan and then queues the request until all previously requested plans complete execution in the order in which they were received. The set state and manage plan message may only be used when no plan is executing and there are no queued plans that are awaiting execution. These two types of messages are sent with different parameter sets, but some of the parameters that they include are used in the same way. We first describe these common parameters and then describe the details of each message.

##### **4.1.5.1. Common parameters**

There are three parameters that the plan request messages have in common in both name and purpose: the *main plan domain*, the *environment-activated actions*, and the *mediated actions*. For both messages, the game provides a domain document describing the operators that can be used in the plan. The main plan domain parameter contains these

operators, which are used by the execution manager to maintain its model of the environment's state as actions are executed during the main plan's execution. They are also used by CrossWind if mediation is performed for the plan. During the execution of a transition plan the operators from the transition domain are used rather than those from the main plan domain; at no time during a plan's execution is the outside-plan domain used. The main plan domain is required with every message requesting that the execution manager manage a plan.

While the main plan is executing, the execution manager allows the game to request permission to perform actions that might not be a part of the plan. The *mediated actions* parameter and the *environment-activated actions* parameter let the game tell the execution manager how these action requests should be handled. Note that these parameters only affect the execution of the main plan, as the execution manager does not process action requests during the execution of the transition plan. Each of these parameters is expressed as an action specifiers document (see Appendix 7.1.1). This type of document describes an action set, which contains zero or more operator names, each of which is paired with one or more sets of object constants for its parameters.

When a step that is ready for execution in the main plan corresponds to an action found in the environment-activated action set, it is not immediately sent to the game for execution. Instead the execution manager holds the plan step until the corresponding action is requested by the game. The game is not informed of the fact that a step is pending and the execution manager will hold the step indefinitely if the game never requests the corresponding action. Generally, actions are included in the environment-activated actions set when their forced execution would break the conventions established by the game. For

example, in most games the user has full control of whether or not his or her character picks up an item that is lying on the ground; the pickup action would be an environment-activated action when it involves a user-controlled character since forcing such a character to pickup an item would break the user's sense of agency within the game world. When the game requests permission to perform an action, the first thing that execution manager checks is its list of pending plan steps; if it finds a pending step that matches the requested action, it instructs the game to execute the action without considering mediation.

If the execution manager instead determines that the requested action is not pending in the plan, it attempts to incorporate the action into the executing plan. This happens even if the action is included in the environment-activated actions set and may occur at some point later in the plan. As described in Section 2.4, reactive mediation is used to ensure that almost any requested step can be executed while also achieving the plan's goals. The execution manager gives the mediated action set to CrossWind so that it can compute mediation responses before and during the execution of the main plan. It is usually necessary for the game to restrict the mediated action set rather than sending CrossWind all actions possible in the domain, otherwise the problem of computing mediation data would be too time-consuming to be reasonable for gaming. If a requested action is not included in the mediated actions set, CrossWind will have computed no responses for that action. In this situation, the execution manager will not know if allowing it to execute will make achieving the goals of the plan impossible. As a result, any requested action that gets to this point in the execution manager's decision process is refused if it is not in the mediated actions set. If it is in the action set, then either the action is consistent with the plan or the execution manager will

have an appropriate mediation response; in either case, the requested action or a substitute will be sent to the game for execution.

In most games, the mediation actions set contains the actions that could be requested by the user and any actions that may spontaneously be requested by the game. This includes actions such as the user pickup action described above. It also includes actions for characters that are controlled by the game to some degree, allowing in-game logic and the execution manager to share control of non-player characters. Finally, actions that change the game world state, such as a change in the weather, may happen based on in-game logic rather than always being initiated by the plan, and would thus be included in the mediated actions set.

The environment-activated actions and mediated actions for one plan request may be different than those for another plan request. This allows the execution manager to be used for general gameplay as well as alternative types of story plans such as a cut scene, in which there are no environment-activated actions and the execution manager has full control of the users' characters resulting in a cinematic user experience. In many cases, actions will be included in both the environment-activated action set and the mediated action set. When an action is included in the latter but not the former, it will always occur as soon as possible when it corresponds to a step in the plan but may also be performed at the request of the game. Since the execution manager and the game could be sending each other messages regarding identical actions at the same time, it is possible that such an action could be executed twice in rapid succession. These types of actions are useful when the in-game logic and the execution manager are sharing control of a character. In the more common scenario where a character is controlled solely by actions corresponding to steps in the story plan, actions relating to that character would appear in neither action set. Finally, an action that is

in the environment-activated action set but is excluded from the mediation action set can only be executed when it both corresponds to a pending step from the plan and is requested by the game. Actions with these characteristics are not used in any of the current Zocalo clients but could prove useful in applications other than gaming or in games with alternative methods of gameplay management. The interaction of the two action sets is shown below in Table 4.1.

Table 4.1 Action set interactions.

<b>Environment-activated action</b>	<b>Mediated action</b>	<b>Description</b>
Yes	Yes	Actions that cannot be forced to occur but are permissible for requests at any point
Yes	No	Actions that cannot be forced to occur and are only permitted to happen when they correspond to steps that are in a plan
No	Yes	Actions that may happen at a game's request but should be forced to occur when they correspond to steps in a plan
No	No	Actions that are not permissible for requests at any point and that will be forced to occur if they correspond to steps in a plan

#### 4.1.5.2. The manage plan message

While the common parameters cover the operator and action components of a request for a plan, the parameters that are specific to each message specify the states and steps of the desired plans. The manage plan message has three additional parameters: the *transition plan goals*, the *main plan goals*, and the *main plan*; of which only the main plan goals parameter is required. For this type of plan request, the game does not specify an initial state for the main plan, but instead provides the transition plan goals parameter which is a goal state document (see Appendix 7.1.4). This document contains a list of literals, each of which is marked as either *true* or *false*. The execution manager is responsible for altering the world

state, if necessary, to make the things that are true in this list also true in the world state and the things that are false in this list also false in the world state. After the execution manager makes the world state consistent with the transition plan goals, it begins the execution of the main plan. The process that the execution manager follows is depicted in Figure 4.1 below.

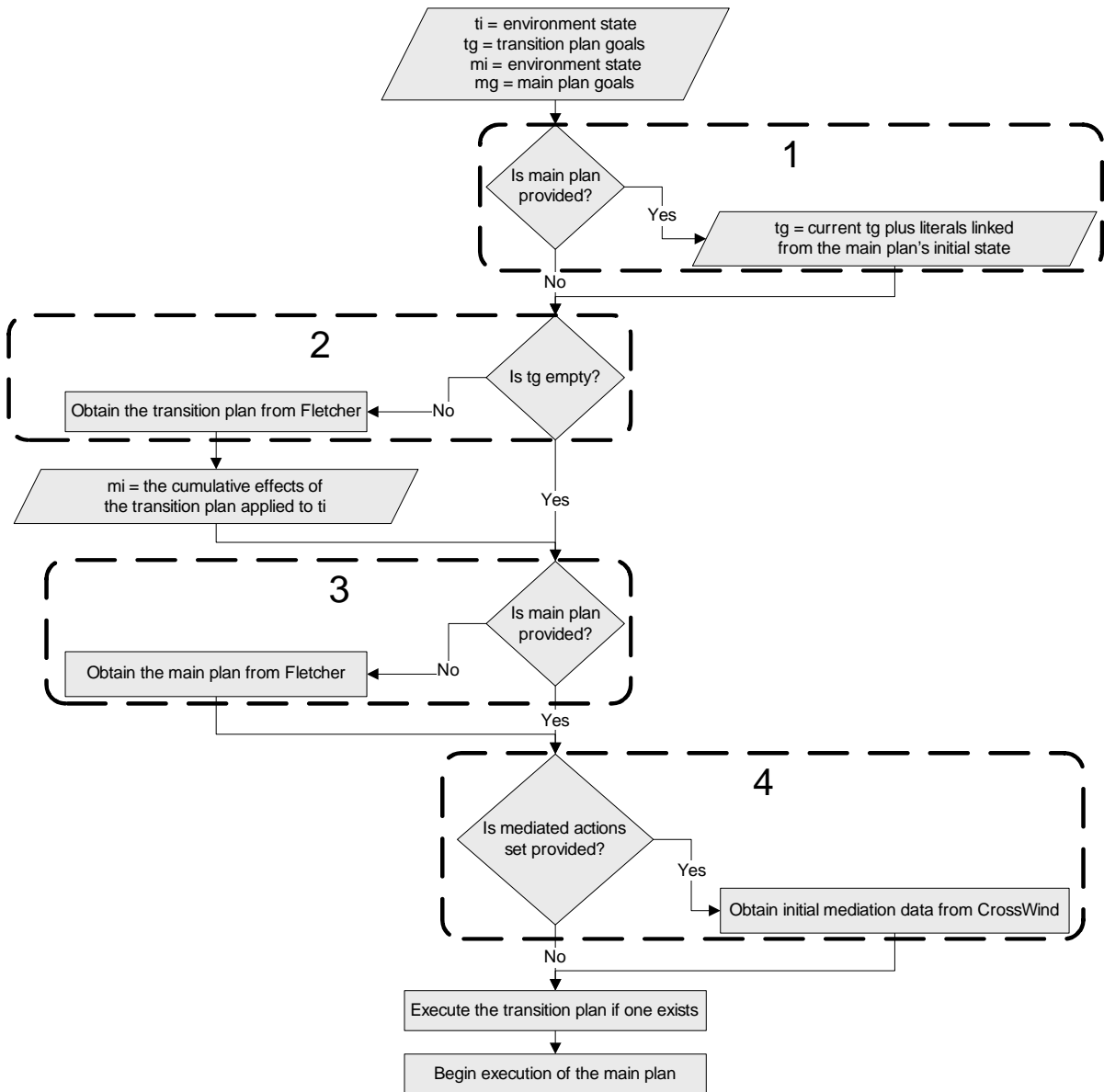


Figure 4.1 Execution manager flow while processing a manage plan message. The variables  $ti$  and  $tg$  make up the planning problem for the transition plan and the variables  $mi$  and  $mg$  make up the planning problem for the main plan. The numbered sections are referred to later in the text.

In some cases, the goals that the game provides with the transition plan goals parameter must be augmented. Since the game may also provide a main plan, the execution manager must ensure that the transition plan goals are consistent with the actual plan that the game provided. If a main plan is provided by the game, then it will contain causal links that

originate at the initial step. These causal links indicate that a step in the plan depends upon a condition being either true or false (according to the truth value of the linked literal) when the plan begins its execution. In order for the execution of the plan to be successful, the truth values of the literals involved in these causal links must match the truth values of the corresponding literals in the world state when the plan's execution begins. To guarantee this, the execution manager analyzes the main plan, extracts these literals, and adds them to the transition plan goals. If there is a conflict, where the transition plan goals contain a literal with a certain truth value and the main plan's initial state has a linked literal with the same predicate and terms but the opposite truth value, then the truth value from the main plan is used. This process of augmenting the transition plan goals is depicted in area labeled with a "1" in the above figure. If the game provided nothing in the main plan parameter, then the transition goals are not changed by the execution manager.

In the case where no main plan is provided and no literals are included in the transition plan goals parameter, there will be no transition plan goals and thus there is no need for a transition plan. In the absence of a transition plan, the execution manager's model of the environment state is used as the initial state of the main plan. In the more common case where there are transition plan goals (regardless of their origin), the execution manager requests a plan from Fletcher by using an initial state that is its model of the environment state and a goal state that is the final set of transition goals. Figure 4.1 depicts this in the area labeled with a "2". The domain that goes with this plan request is the transition domain that was provided by the game earlier as discussed in Section 4.1.4. The complete plan returned by Fletcher is used as the transition plan.

To prevent any delay between the execution of the transition plan and the execution of the main plan, the execution manager does not begin the execution of the transition plan until it also has everything that it will need for the execution of the main plan. In order to begin execution of the main plan, the execution manager needs the main plan itself and an initial policy table from CrossWind if the mediated actions set is not empty. If the main plan was provided as a parameter then the provided plan is used, otherwise the execution manager obtains the main plan from Fletcher (shown in the area labeled with a “3” in the figure above). The request that the execution manager sends to Fletcher includes the main plan goals and the main plan domain, as specified in the manage plan message parameters. It also must contain the initial state of the main plan. Since the execution manager has a transition plan including the full initial state, it can compute what the state of the game world will be at the end of the transition plan’s execution, which is also the initial state of the main plan. The execution manager computes this state by first making a copy of its model of the environment’s initial state. It then alters that state with the effects of the steps in the transition plan, applying their effects in the order in which the steps’ corresponding actions will be executed. After the effects of transition plan steps have been applied, the state model will be exactly the same as the world state after executing the transition plan. The execution manager uses this as the initial state of the main plan when sending its request to Fletcher.

If the manage plan message included actions in the mediated actions set then the execution manager will also obtain an initial policy table from CrossWind before executing the transition plan (shown in Figure 4.1 in the area labeled with a “4”). The execution manager’s request to CrossWind includes the mediated actions set and the main plan along with its initial state and goal state. If a main plan was provided by the game, the initial state

used here may not be exactly the same as the one used to generate the plan. The initial state that is used for mediation (which is the world state after the transition plan's effects are taken into account) will be consistent with the initial state that was used in creating the provided main plan to a certain extent; all of the initial state literals that were used by steps in the plan will have the same truth values in both initial states. If the game needs to specify an exact initial state then it must use the set state and manage plan message rather than the manage plan message.

Having collected a transition plan (if a transition is needed), a main plan, and mediation data for the main plan (if mediation is needed), the execution manager is ready to respond to the plan request. It first instructs the execution environment to execute the actions that correspond to the steps in the transition plan and then the actions that correspond to the steps in the main plan. The details of plan execution are discussed in Section 4.1.6. After the execution of the main plan is complete, the execution manager returns to its outside-plan behavior unless other manage plan messages were received during the processing and execution of the plan request that just completed.

If the game sends one or more additional manage plan messages during the processing of a previous plan request, those requests will be queued and their plans executed in the order in which the requests were received. Upon receiving a manage plan request during another plan's execution, the execution manager goes as far as possible through the flow of events depicted in Figure 4.1. This means that it can continue processing the request until the environment state is needed because the final environment state at the end of the executing plan will very likely be different than the environment state during that plan's execution. In the figure above, the operations that the execution manager can perform

beforehand are depicted along the vertical line running down the center. Once the execution manager's flow must branch off to a side, it must queue the plan request and only resume processing it after the preceding plans have executed completely. If the execution manager is able to reach the bottom of the diagram during this preprocessing, it will queue the main plan rather than begin its execution in the middle of another plan (if the execution manager went this far then there cannot be a transition plan since it did not take the branch in the area labeled with a "2" in the figure).

#### **4.1.5.3. The set state and manage plan message**

When the message from the game instructs the execution manager to set its model of the environment's current state and also begin managing a plan, the parameters that can be used to specify the states and steps of the plan are different than those for the manage plan message. In addition to the common parameters, the two parameters that the game may include with the set state and manage plan message are the *main plan* parameter and the *main plan problem* parameter. The main plan problem parameter is a problem document (see Appendix 7.1.10), containing the full specification of an initial state along with a goal state; it is required with every set state and manage plan message. In processing this message, the execution manager extracts the initial state from the main plan problem and uses it to overwrite its model of the environment's current state. It is up to the game to alter its world state so that it is exactly the same as the initial state when the execution manager receives this request for a plan. The execution manager will reject this type of message if it is executing another plan or a previous plan request is pending; to indicate this failure to the game, it sends the action refused message with no request data and an error code indicating that the environment state cannot be set during the execution of a plan. This type of plan

request is commonly used when a game is altering its state as game play starts in a newly loaded game world map or when a previously saved game is restored.

If the optional main plan parameter contains a plan then the execution manager will use that plan as the main plan for this plan request. The game is responsible for ensuring that the main plan problem is consistent with the provided main plan. If the game does not provide a main plan, then the execution manager uses the main plan domain and the main plan problem to request a complete plan from Fletcher. The plan that Fletcher computes is used as the main plan in this case. In this type of plan request, there is never a transition plan. Once it has the main plan (regardless of the source), if the game provided actions in the mediated actions set then the execution manager contacts CrossWind requesting an initial policy table. Finally, the execution manager begins the execution of the main plan.

#### **4.1.6. Plan execution**

Once the execution manager is ready to begin executing a plan (whether it be a transition plan or a main plan), it instructs the execution environment to perform the actions that correspond to the plan's steps. Each action is sent as soon as its planner-imposed ordering restrictions are satisfied. The actions are sent with the perform action message, described in Section 4.1.3, always using an empty request data parameter. In order to quickly determine which steps are ready for execution, the execution manager constructs a directed acyclic graph (DAG) before it sends any actions to the execution environment. Each step in the plan is represented as a vertex in the DAG. For every pair of steps,  $a$  and  $b$ , where there is an ordering link that orders  $a$  before  $b$  in the plan, the execution manager adds an edge to the DAG that starts at the vertex representing  $a$  and goes to the vertex representing  $b$ . After an action has executed successfully the corresponding vertex is removed from the DAG along

with all of the vertex's edges. Since each causal link implies an ordering link, this results in a DAG where a step is ready for execution if and only if it has no incoming edges.

When the DAG is first constructed, generally the vertex representing the plan's initial state will have outgoing edges to every other vertex and the vertex representing the plan's goal state will have incoming edges from every other vertex. This is always the case for plans constructed by Fletcher, but the execution manager could also handle a handcrafted plan with a different structure as long as it contained no ordering cycles among its steps. Once the DAG is constructed, the execution manager sends a perform action message for the each action corresponding to a step that is ready for execution as long as it does not match any of the actions contained in the environment-activated actions set. At first, this will only be the initial step, for which the is initial step parameter in the perform action message will be set to *true*. Each perform action message includes a unique action identifier. When the execution manager receives an action succeeded message from the game with an action identifier that corresponds to an action that was generated from a plan step, the execution manager removes the corresponding vertex from the DAG. After the removal of a vertex, the execution manager finds any remaining vertices that no longer have any incoming edges and repeats the process of sending perform action messages.

When the execution manager encounters an action that is ready for execution but is contained in the environment-activated actions set, it does not send the perform action message to the execution environment. It records the step corresponding to the action as pending until it receives a new action request message that matches the pending step. When a matching new action request is received, a perform action message is sent in response with an action identifier that the execution manager associates with the vertex and the request data

from the new action request message. If the game never requests an action for a pending step, then that perform action message will never be sent and the plan's execution will not complete. Work in the areas of temporal planning, plan recognition, and proactive mediation may lead to a response for this situation that will result in the successful completion of such a plan [8]. However, for this work the execution manager will wait for the user indefinitely. This is discussed further in Section 5.2.

Eventually, the vertex representing the plan's goal step will be the only one that remains. The perform action message sent for this vertex will have its is goal step parameter set to *true*. For all actions in the transition plan, even the ones corresponding to the initial step and the goal step, the is transition action flag sent with the perform action message will be set as *true*. When the DAG has no remaining vertices, the execution manager attempts to begin the next plan, which may be a main plan following a transition plan or may be a plan from a queued manage plan message. If a plan is waiting, then a new DAG is constructed for it and the plan execution process restarts from the beginning. If the plan is a main plan, then all of the resulting perform action messages will have the is transition action flag set as *false*. When there are no more queued plans, the execution manager enters its outside-plan state.

#### **4.1.7. Adding requested actions to an executing plan**

During the execution of a main plan, the game may still send new action request messages to the execution manager. The way in which the execution manager incorporates a requested action into the plan was briefly discussed in earlier sections; here the process is described in more detail. When the game sends the execution manager a new action request message, the execution manager will respond with either a perform action message or an action refused message. If there is no plan executing then the execution manager responds

with a perform action message as described in Section 4.1.3. If a transition plan is executing then the execution manager responds with an action refused message with parameters containing an error code indicating that the executing plan does not support requested actions.

When a main plan is executing and a new action request message is received, the execution manager first checks the list of pending plan steps that were held as environment-activated actions. If the request action matches one of these steps then the execution manager sends the environment a perform action message for the requested action. When the action executes successfully and the game sends the execution manager an action succeeded message, the execution manager removes that step's corresponding vertex from the DAG and continues with the plan's execution as described in Section 4.1.6.

If the requested action does not correspond to a pending plan step then the execution manager attempts to incorporate the action into the plan through mediation. If no mediated actions set was provided with the original plan request, then the execution manager responds with an action refused message containing an error code indicating that no mediation data was available. If the game did provide mediated actions with the plan request then the execution manager will have a policy table for the executing plan. During the plan's execution, the execution manager keeps track of which mediation responses are valid since some can only be performed between certain steps in the plan. The execution manager checks its list of actions that need responses for the current point in the plan and, if there is an entry for the requested action, then the execution manager knows the requested action is exceptional to the plan; otherwise it knows the requested action is consistent with the plan. In the case of a consistent action the execution manager instructs the environment to perform

the requested action. Since the policy table is not updated to take the effects of this action into account, it is possible that some of the responses for future exceptional action requests may no longer be valid. This is a limitation of CrossWind, as it does not provide a way for this information to be incorporated into the mediation process; we discuss this limitation further in Section 5.2. In the case where the requested action is exceptional, the execution manager examines the list of responses and chooses the first one that is still valid given the effects of previously requested actions that were allowed to execute. If none of the responses can be used, then the execution manager sends an action refused message to the environment with an error code indicating that no acceptable responses could be found. In the cases where an acceptable response does exist in the policy table, it will be either an intervention or an accommodation. This process of determining a response to an action request is depicted in Figure 4.2.

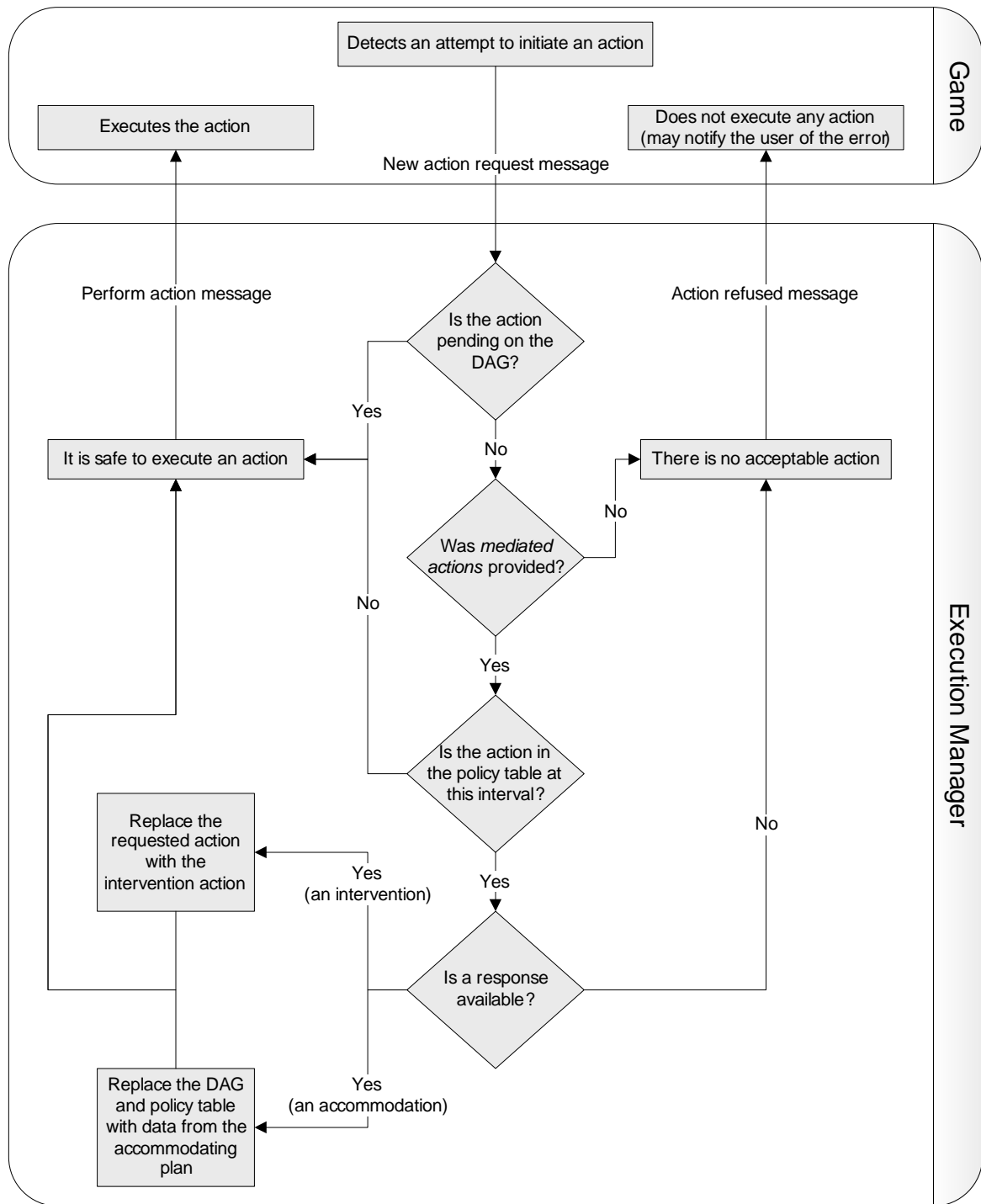


Figure 4.2 Execution manager flow while processing a new action request message during plan execution. Beginning where the game detects that an action has been initiated, this diagram depicts the decision process used by the execution manager in determining how to respond to the new action. This figure only depicts the process that occurs when the execution manager is managing the execution of a main plan.

If the chosen response is an intervention, the execution manager sends a perform step message to the environment but it replaces the requested operator name and bindings with those found in the intervention. By doing so, the game will execute an action that only has effects that do not conflict with the remainder of the plan and that a game developer has designated as an acceptable substitute for the requested action. If the chosen response is an accommodation then the execution manager sends the execution environment a perform action message with the requested parameters and it obtains the accommodation's policy table and plan from CrossWind. The execution manager's policy table only contains an accommodation if the replacement policy table and plan have already been computed by CrossWind so there will be no delays in obtaining this data other than the time it takes for the messages to travel to and from CrossWind. The execution manager updates its internal data structures when it receives the new plan and policy table. While it is waiting for CrossWind's response and updating its data structures, the execution manager does not process any new action request messages that it receives. As soon as the new policy table and DAG are ready, any action requests that have accumulated are processed. This limitation of the execution manager is discussed further in Section 5.2.

#### **4.1.8. Features for game development**

In addition to the messages described in the previous sections, the execution manager supports a small range of debugging features. If the game sends the *register EM remoting service* message, then the execution manager may be simultaneously accessed by applications other than the game responsible for its instantiation. Once a connection is made using an additional TCP channel and .NET Remoting, the remote client may send the execution manager the same messages as a game sends. The additional application may also

add its own implementation of the execution environment interface to the execution manager's list of execution environments. If the execution manager is given multiple execution environments, then they are all notified of perform action messages and action refused messages as they are sent by the execution manager.

The only application that currently takes advantage of this additional execution manager connectivity is a demonstration and development tool named the *action executor*. This simple Windows application presents its user with a graphical user interface for sending messages to the execution manager using XML files loaded by the user and data typed into the application's interface. During demonstrations, users may adjust the state of the world before requesting a plan in order to see the execution manager adjust the transition plan as necessary. Furthermore, the ability to execute individual actions without the need for a complete plan is useful for debugging the in-game code responsible for causing the effects of those actions to occur in the game world. An example follows, describing how this in-game code is structured in one execution environment.

## **4.2. An example execution environment implementation**

The distinction between what is part of Zocalo and what is outside of its scope comes at the boundary between the game and the execution manager. The execution manager interface is part of Zocalo and is actually defined in the execution manager's .NET assembly. The requirements placed on the game, such as the need to send a new action request message before executing any action, are also within the scope of Zocalo. Concrete implementations of the execution environment are not a part of Zocalo, but we describe one here since having an understanding a typical implementation helps in understanding Zocalo as a whole.

The implementation of the execution environment in Unreal Tournament 2004 (UT2k4) builds upon modifications made in Unreal Tournament game engine during work on the Mimesis architecture [23]. We have also implemented the execution environment in the open-source game engine *Source*, which is based upon the same technology used by the popular Half-Life 2 game. The Half-Life 2 and the UT2k4 versions of the execution environment are very similar. They both use the concepts of *class* and *object* in the object-oriented programming sense, but implementations using other programming paradigms could also be appropriate, especially in existing game engines that do not use an object-oriented approach for their own source code.

In the UT2k4 developer community, large extensions to the game engine are called *mods*. We have developed a Zocalo mod for UT2k4; our mod is the implementation of the execution environment. We use TCP/IP to communicate with the execution manager in its EM-SS form. This choice is necessitated by the fact that the source code and header files for UT2k4 are not publicly available. Our mod is developed using UnrealScript, a Java-like language that provides access to many functions of the UT2k4 game engine. UT2k4 comes with UnrealScript classes that facilitate network communications, but it lacks built-in support for XML. We implemented a simple XML parser that can process the XML fragments sent by the EM-SS. Some of the messages sent to the execution manager are assembled in the game at runtime (e.g., action requests) and some of the more complicated messages are loaded from text files, which are usually authored well before the game is executed.

#### **4.2.1. Starting a plan-driven game**

When a player begins a Zocalo gaming session using UT2k4, they do so by starting the Zocalo mod. This also starts the EM-SS in the background. After starting UT2k4, the

player chooses a map file, which was created earlier by a Zocalo mod developer. The map file contains information about world geometry and objects in the game world; it is also linked to a file containing the XML for a set state and manage plan message, with an initial state description that matches the state of the game world stored in the map file. The game creates an UnrealScript object, called the *ZLink*, that listens for XML fragments from the execution manager until the user exits the game. It then loads the world map, connects to the execution manager, and sends it the set state and manage plan message. When the game receives a perform action message from the execution manager, it will be the action representing the main plan's initial step. The game constructs the XML fragment for an action succeeded message and sends it to the execution manager. The game now allows the player to begin interacting with the virtual world.

#### **4.2.2. Executing actions**

In the UT2k4 Zocalo mod, the code for each action that can occur in the game world is organized into its own class. For each action, a game developer defines an *action class*, a class definition that specifies the behavior of an action operator represented within one of the domains for that game. Methods within each action class perform checks for the action's applicability to the current game state, run the code responsible for the state change associated with the action, and verify that the action, once executed, has modified the game state as expected. When the *ZLink* receives a perform action message, it sends the message to the *ZActionExecutor* class. The *ZActionExecutor* uses a lookup table to map from the action's operator name to the name of the action class that is the action's implementation. It also uses a lookup table to change the strings given as the action bindings into references to the corresponding game objects. Having performed these translations, the *ZActionExecutor*

creates an instance of the action class and calls its *CheckPreConditions* method. This method, different for each action class, returns *true* if the world state is such that the action can be executed successfully, or *false* if the world state makes executing the action impossible. If the return value is *false* then the *ZActionExecutor* sends an action failed message to the execution manager. In general, this only happens during development, if there is a programming error in the action class or there is a logic error in the domain. If the return value is *true*, then the *ZActionExecutor* calls the *Execute* method on the action class instance, with the binding references as parameters.

The *Execute* method of the action class performs the animations and world state changes that are dictated by the semantics of the corresponding operator. For example, in a walk action, the action class uses UnrealScript functions to find a path from the character's current location to the character's destination. It exercises the path-finding features built into UT2k4 and may use intermediate locations as target destinations along the way to the final destination specified by the action parameters. The action class is finished with the execution of the action, it calls its own *CheckPostConditions* method, which returns *true* if the effects have been achieved, or *false* if there was a failure. The action class calls the *ZActionExecutor's ReportActionExecuted* method, passing it the result of the *CheckPostConditions* method. If the value is *true* then the *ZActionExecutor* sends the action succeeded message; if it is *false*, it sends the action failed message. Again, this false result should only occur if there is a programming error in the action class or a logic error in the domain. These errors should be eliminated during game development.

### 4.2.3. Detecting user actions

When the user interacts with the world or when a world event is triggered by a game-controlled character, these actions must be approved by the execution manager before executing. In UT2k4, objects in the world are represented by various classes, such as *Trigger*, *Player*, and *Pickup*. Methods of these class instances are executed by the game engine when an event occurs in the world, such as a player touching a weapon lying on the ground (called a *pickup* in UT2k4). The Zocalo mod for UT2k4 overrides the default behavior for these methods. Rather than adding the weapon to the player's inventory, the modified version of this function creates an action class instance corresponding to an operator named *Pickup\_Weapon*, using strings for the bindings that correspond to the parameters of the action. The Pickup object that was touched sends this action class object to an UnrealScript object called the *ZMediationInterface*. This object sends a new action request message to the execution manager and then terminates. This means that the pickup action does not occur in direct response to the event that triggered it. The execution manager processes this action request as described in Section 4.1.7. If no mediation is required, ZLink object will receive a perform action message from the execution manager instructing it to execute a *Pickup\_Weapon* action with the same bindings. In the case where mediation is required the ZLink may receive a perform action message with a different operator name and bindings or it may receive an action refused message. It handles perform action messages as described in the previous section and it does nothing in response to action refused messages. The response time from the execution manager is generally short enough that the user does not perceive any delays while playing the game. As mentioned earlier, the policy of doing nothing in

response to refused actions is not always an ideal solution but we have yet to develop a generally acceptable strategy (ideas are discussed in Section 5.2).

#### **4.2.4. Proceeding after the end of the plan**

After all of the plan steps have been executed, the ZLink will receive a perform action message with the *is goal step* set as *true*. The ZActionExecutor sends the execution manager a corresponding action succeeded message and allows the user to continue exploring the game world. User actions will still be routed through the execution manager, but no mediation will occur and no actions will originate from the execution manager. In order to experience another story plan, the user can either load a new map or choose another story file. The chosen story file must have been created by a game developer to be compatible with the actions supported by the loaded map. For example, a story file that describes goals related to an avalanche can only be used in a map that contains a mountain with precariously placed rocks or some similar scenario. This story file has different contents than the one linked to the map when it was originally loaded; it contains the XML fragment for a manage plan message, meaning it contains a domain, transition plan goals, main plan goals, and possibly a pre-computed main plan, an environment initiated actions set, and a mediated actions set. The game sends this message to the execution manager using the ZLink. After the execution manager has prepared the plans, the ZLink will receive a perform action message for the first initial step. If the execution manager is using a transition plan then the *is transition action* flag will be set as *true*. The game executes all of the steps from the transition plan and then the main plan begins and executes in the same manner as we described above.

### 4.3. The Fletcher Web service implementation

All of the plans used by the game and the execution manager are created by Fletcher. Fletcher is a Web service that accepts messages sent over HTTP, encoded in SOAP (introduced in Section 2.2). Its interface is expressed in WSDL, which allows application developers to automatically generate a client proxy in many programming languages. The execution manager is a Fletcher client that only uses Fletcher in the simplest way, by requesting that it provide the first complete plan for a planning problem. Fletcher also supports more complex interactions that are used by CrossWind and often used by game developers with the help of an application called *Bowman*, which is described at the end of this chapter.

Fletcher accepts five different types of messages, each requesting a different operation related to the planning process. As described in Section 2.1, planning contexts are the basic objects upon which Fletcher and its underlying planning system operate. Clients can send messages to Fletcher requesting that it

- create planning contexts
- respond with information about a planning context
- expand the explored search space of a planning context
- save and load planning contexts to and from a database
- provide a list of the names that identify the heuristic search functions available to the planning system

Bundled together, the first three types of messages compose the functionality of a typical planning system. It is common for a planning system to provide a single function (or at least, a small number of functions) to access this functionality. Fletcher accepts a number

of different message types that accomplish those tasks in different ways, giving the user relatively specific control over the tasks' performance. The following sections further explain how these messages are used to control Fletcher.

### **4.3.1. Creating a planning context**

The first step in using Fletcher to compute a solution to a planning problem is for a client to establish a planning context. One message type used to establish a context contains a domain, a problem, and the name of a heuristic search function. Fletcher uses the operators in the domain to create new plans during the refinement search process. Fletcher uses the problem to construct an empty plan with appropriate initial and goal states. As described in Section 2.1, this empty plan forms the root node of the plan space associated with the planning context. The specified heuristic is used to guide the planner's search through the plan space of this context. The heuristic name that the client provides must match one of the heuristics that exist on the server hosting Fletcher (a message for retrieving the valid heuristic search function names from a Fletcher service is described in Section 4.3.5).

When Fletcher receives a message, it uses the corresponding XML schemas found in the planning utilities assembly to check that the input is valid. If it is not valid, then Fletcher responds with a SOAP fault, explaining why the request was not processed. If the input data is valid, Fletcher creates a new planning context and sends the client a response that contains an identifier to be used to refer to the new planning context. By using the session cookie and the planning context identifier in future requests, the client can work with that planning context for the duration of the session. Clients can create multiple planning contexts by sending more than one of these requests; for each request, Fletcher returns a new planning context identifier.

A second message type used to establish a planning context includes slightly different arguments. In this message type, a client sends a domain, a partial plan specification, and the name of a heuristic search function. The partial plan specification must come in the form of a plan node document, which is defined by a schema in planning utilities (see Appendix 7.1.6). Fletcher establishes a planning context as in the prior message type's case, but rather than use a problem to create the root node of a plan space, the partial plan specification is used to create a root node already containing plan structure. This message type is used to achieve the "seeding" approach to refinement search discussed in Section 2.1.

Fletcher provides a number of variations on these two basic messages; they serve as a convenience for the client and help to minimize the amount of data that must be transmitted between the client and the server. One option is for clients to send a domain document as well as either the problem document or a plan node document; this XML must conform to the corresponding schemas. Alternatively, clients may first compress the XML using *gzip* and send the Base64 encoding of the resulting binary data, reducing the size of the transmitted data (usually resulting in a message that is one tenth of the uncompressed size). Finally, clients may provide planning context identifiers referring to planning contexts created by the same client earlier in the session. Fletcher will copy the indicated information from those existing planning contexts. By specifying a combination of new information and references to information already established in other planning contexts, a client is able to send less data when creating planning contexts. All of these possibilities are specified in the planning context document schema (see Appendix 7.1.5), which is the data type used for this Fletcher message.

### **4.3.2. Retrieving planning context information**

At any point after creating a planning context, the client may request information about the context's state. Fletcher accepts requests both for information about the context's plan space as a whole and for information about a particular plan node in the plan space. When requesting information about an entire plan space, the client only specifies the planning context identifier in the request. Fletcher responds to this request with a plan space document (see Appendix 7.1.7) describing the tree of plan nodes that the planner has constructed during plan space exploration, explained in the next section. Each plan node that this document describes is identified by a plan node ID value. The plan space description also contains a ranking value for each plan node. This ranking value is computed by the planning context's heuristic search function. When requesting information about an individual plan in the plan space, the client specifies both the planning context identifier and the plan node ID in the request. Fletcher responds to this request with the corresponding plan node document.

### **4.3.3. Exploring a search space**

After creating a planning context, the client can instruct Fletcher to explore the search space in order to find a solution to the context's planning problem. Two types of messages exist to initiate this activity, both of which have more than one variation.

One type of message is a request to refine a particular node in the context's search space. The client must provide a plan node ID to identify the node to be refined. As a convenience, Fletcher also accepts a message indicating that it should refine the plan node on the fringe of the plan space that is currently ranked as the best plan by the context's heuristic search function. Fletcher responds to the client with a message containing the ID of the plan node that was refined.

The other type of message is a request to repeatedly refine the best-ranked plan nodes on the fringe. This type of operation continues until a termination condition is met. The client can tell Fletcher to halt its search a) as soon as it finds a complete plan, b) once it has refined a certain number of nodes, or c) once a certain amount of processor time has transpired on the server. If the client requests that Fletcher halt as soon as it finds the first complete plan, then Fletcher will either return the ID of the plan node representing the complete plan that it finds or an error value if it cannot find a complete plan. This error condition occurs if Fletcher searches every node in the plan space, if its search exceeds the given time limit, or if its search exceeds a server-specific resource bound. If the client requests that Fletcher expand the search space for a certain number of refinements, then Fletcher will respond with the number of nodes that it actually refined, which may be less than the requested number if the time limit was reached or if the search space was exhausted.

#### **4.3.4. Using the database**

If a client application is using Fletcher based on the interactive input of a user, it may need to end the planning session and come back to the same planning problems at a later time. Fletcher allows clients to save planning contexts to a database and to load the saved progress in later sessions. Save and load requests both require a message header that holds username and password parameters; Fletcher only allows a client to load a planning context if it supplies the same username and password as was given when that planning context was saved. In our current implementation, the Fletcher server must know the username and password before it can be used by a client. Fletcher's current techniques of authentication and authorization are very simple; they should be extended before Fletcher is used on a broad scale.

The request to save a planning context contains the planning context identifier as input; the response from Fletcher is the ID of the new database entry. The request to load a planning context contains the ID of a database entry as input and the response from Fletcher contains the new planning context identifier that specifies the loaded planning context. Note that a planning context identifier is only meaningful during a single session while the ID of an entry in the database refers to the same entity irrespective of the session.

#### **4.3.5. Retrieving heuristic search function information**

When creating a planning context, clients must specify the name of the heuristic search function that should be used during the planning process. The name that the client specifies must match the string name of a search function that is implemented on the Fletcher server. Clients can send a message requesting a list of the available heuristic functions' names and Fletcher will respond with an array of strings. The client can then use any of those strings when sending a request to specify the use of a heuristic search function when creating a new planning context. The client is not required to include session information with this request and the response will not include session information.

In order to change or add heuristics to a Fletcher deployment, Fletcher must be recompiled with the source code for that heuristic. This means that only developers running their own Fletcher server can add heuristics. Future work on heuristic representations could allow Fletcher clients to provide descriptions of their own heuristic search functions; which Fletcher could then use without the need for redeployment.

#### **4.3.6. Implementing a Fletcher client**

Implementing a Fletcher client that sends requests and receives responses as described above is made easier because of Fletcher's use of SOAP and WSDL. Since

Fletcher uses these industry standards, developers can leverage existing development tools when implementing a Fletcher client. These tools generate code for C#, VB.NET, or Java based on a WSDL document. Some available tools are:

- Web Services Description Language Tool (a Microsoft tool for .NET),
- WSDL2Java (an Apache Axis tool for Java), and
- WSContractFirst (a thinkecture tool for .NET).

After using one of these tools to generate client code, application developers can write an application that sends messages to Fletcher and receives responses by simply calling methods on a local object. These same tools can be used to generate client code for accessing any Web service that uses WSDL and SOAP. In the following section we describe extensions to Zocalo, which include Web services that support these standards and a client application that uses client code generated by the tools listed above.

#### **4.4. Implemented extensions to the Zocalo architecture**

Zocalo is based upon the principles of service-oriented computing and, in particular, it is designed to allow for the addition of new applications and the reuse of existing applications in new ways. The applications described in the previous sections are central to Zocalo but additional applications, which provide functionality not included in the core Zocalo implementation, have already been implemented as parts of other work. These extensions will likely be followed by more in the future. We consider an application to be a part of the Zocalo architecture if it allows integration into other applications via a public interface that uses the data formats defined in the planning utilities assembly. The functionality made available by these new applications can be used directly by Zocalo clients or by other applications within Zocalo. We describe briefly these additional applications and

discuss how each makes use of Zocalo's service-oriented architecture to add new functionality.

#### **4.4.1. Mediation Web services**

If users are allowed to interact with the execution environment during the execution of a story plan, there is a possibility that a user action will disrupt the plan, making further plan execution impossible. The techniques of reactive mediation (in [17]) and proactive mediation (in [6]) have been developed to allow successful plan execution in the face of user interaction. Reactive mediation, as implemented in CrossWind, is described in Section 2.4 and in Section 4.1.7 on the execution manager above. By augmenting the concepts of reactive mediation with a plan recognizing component, proactive mediation takes the approach of computing alternative plans before an exceptional action makes them necessary. By using predictions about what actions a user will perform, proactive mediation is able to integrate the user's future actions into the plan before they occur in the execution environment. The *Kyudo* Web service is an initial implementation of proactive mediation. When the systems that provide it with input are further developed, *Kyudo* will easily be integrated into the execution manager in a later endeavor. Proactive mediation uses information about predicted user actions to substitute an accommodating plan before the user actually makes an exceptional action request. We further discuss the benefits of proactive mediation in Section 5.2.

#### **4.4.2. The Bowman planning client**

While game execution is the focus of the core Zocalo applications, game development is facilitated by the Zocalo client named *Bowman*, which uses the core Zocalo applications. *Bowman* provides developers with a graphical interface to Fletcher, allowing

them to specify and refine operator libraries, adjust search control rules and define ending world states (or story goals). As the developers change these inputs, Fletcher creates new plan spaces that are returned to Bowman. Developers can review these plan spaces and can adjust the input to the planning contexts accordingly. The Bowman Zocalo client is an example of direct use of Zocalo's Web services for a purpose related to, but separate from, that of the execution manager.

## **5. DISCUSSION**

In the previous chapters, we describe the current implementations of the Zocalo applications, but there are extensions that could be made to enhance existing Zocalo applications in future versions. Some of these suggested enhancements add new features that enable new usage scenarios and others address limitations in the current implementations. These enhancements are organized into two sections: first, those applying to Zocalo Web services, followed by those applying to the execution environment and the execution manager. Finally, we briefly summarize the work presented in this thesis.

### **5.1. Future work related to the Zocalo Web services**

While Fletcher works well for the most common usage scenarios, its feature set could be enhanced to allow new types of interactions. The first enhancements address Web service technologies and can be applied not only to Fletcher but also to CrossWind, Kyudo, and any future Zocalo Web services. The rest of the enhancements are focused specifically around Fletcher.

Zocalo Web services are made accessible to application developers through the use of WSDL and SOAP, but these are just the first steps towards making the creation and use of a Zocalo client a predominantly automated task. The Universal Discovery, Description, and Integration (UDDI) standards are one example of a technology that could be used to enhance the Zocalo Web services [13]. If the Zocalo Web services were deployed on more servers, they could be listed in a UDDI repository along with the WSDL Web service descriptions. This would allow a client to discover and use the Web services without prior knowledge of individual servers. In addition, UDDI structures information that can help a client determine

which server is most appropriate when more than one is available. Having a variety of available servers would be essential in order to bring Zocalo out of the research lab and into a more commercial setting.

To spur on wider use of Fletcher, authentication of users is one feature area where improvements would be wise. Allowing users to create server accounts and control access to the planning data that they create will be a step towards building a large repository of planning problems and solutions. Applications for collaborative plan space analysis could use Fletcher as a uniting bond between game developers and AI researchers in different geographical locations. To support this, Fletcher could be extended to allow users to share planning contexts with others. In addition to Fletcher clients, the Fletcher Web service could use this public planning data when exploring new planning contexts. It may be possible to reduce the time needed to respond to client requests by finding complete plans using a case-based approach like the one used by Hammond [4]. Since it is designed to be accessed remotely, Fletcher is well suited for extensions in this area; it could collect planning problems from a variety of users and store them in a persistent database along with any plans it computes. Clients that use this enhanced version of Fletcher would be able to benefit from the case-based planning approach without locally storing any voluminous data from previous planning problems.

An improvement that would likely come with work in extending Fletcher for case-based planning, plan space summarization and more detailed descriptions of plan nodes would help developers using Fletcher to iteratively explore planning contexts. This data could be added to the planning utilities schemas for use by developer tools like Bowman. Even if case-based planning is not the driving force behind such improvements, Bowman and

possibly other Zocalo clients would benefit from enhancements to the planning utilities schemas in the form of additional information about planning structures.

As mentioned earlier, improvements to Fletcher's support for heuristics could make its search of planning contexts more effective. Currently heuristics must be expressed as .NET classes that are compiled with Fletcher, but a possible extension to Fletcher would allow clients to specify their own heuristics. One approach to achieve this functionality is to define a language that Fletcher clients could use to describe a large family of heuristics. This would allow experimentation with heuristics that vary along commonly used dimensions. Another possibility would be to allow users to provide C# source code, which Fletcher would then validate and dynamically load. This would allow clients to dynamically use the full range of heuristics supported by Fletcher's underlying planner.

## **5.2. Future work related to plan execution**

In the execution environment and the execution manager, there are some plan execution situations that lead to less than desirable results for the user playing the game. We suggest solutions for some of these situations; many of which were also issues with the Mimesis architecture.

When specifying the initial state in a set state and manage plan request, the game uses the closed world assumption to implicitly define the truth values of all of the false initial state literals. This allows the game to avoid enumerating many literals. In the situation where the game is using the manage plan message, it can only use a goal state document to specify the transition plan goals. This means that in order to guarantee an exact world state to the same level of detail as an initial state specification, the game must enumerate all of the literals, both true and false, which is usually not practical. To give the game more control over the

main plan's initial state with the manage plan message, Fletcher could be extended to support the closed world assumption for goals. This would extend the planner's definition of a flaw, introduced in Section 2.1, to include situations where the implicit goal literals are not satisfied. With this extension, the execution manager could be used to build a transition plan that uses as its goals the full initial state of a pre-computed main plan. This extension would also be the first step towards allowing an even more powerful way of specifying the parameters of a plan request. The second extension would allow the game to choose literals from the current state and use their truth values to override those of a goal state treated under the closed world assumption. The game developer could write this list of literals at design time and the runtime truth values would be used for the initial state of the main plan.

During plan execution, the process of mediation can lead to various delays in action execution. When accommodation is used, the policy table and the execution DAG must be updated with those from the accommodation response. This requires some processing time and could introduce a delay in action execution if either of these data structures is large. In addition, the current implementation of the execution manager must contact CrossWind to retrieve the accommodating plan and policy table after the user has requested the action. Any actions that are requested during this process are delayed until the execution DAG and policy table have been updated. Both of these issues could be addressed by caching and preprocessing accommodation data, but the execution manager must strike a balance between loading too little data and loading too much data. Attempting to work too far ahead could put a heavy load on the machine's resources, reducing the game's performance, but loading too little data could result in delayed perform action messages.

Increased communication with CrossWind would also allow the execution manager to update its policy table after allowing the execution of consistent actions and actions from interventions. As these actions are executed, the state of the world can become different than the state modeled by CrossWind. The CrossWind implementation does not have a feature that allows the execution manager or other clients to inform it of these changes. As a result the order of the responses in the policy table may not be ideal. One situation where the ordering of responses may change occurs when a failure mode is repeated multiple times. Choosing a different failure mode may be considered better by CrossWind's heuristic to prevent the user from growing suspicious of whether they can really affect change in the world. If the execution manager informed CrossWind of all actions as they executed, CrossWind could keep this policy table up to date and would also be more effective in its search for accommodations.

Environment-activated actions can also present complications in some situations. If the main plan includes a step that corresponds to an environment-activated action, then there is the possibility that its action will never execute. By virtue of it being environment-activated, the execution manager cannot force the action onto the game, but in many cases these actions are only executed based on user input. If the user does not provide the input to trigger the action, then the execution of the plan can come to a standstill. However, temporal reasoning is under development for Zocalo and provides hope for these situations [8]. One solution would be to attach an abort time to environment-activated actions. The execution manager would begin a timer when the action is added to its list of pending actions and when the timer expires, if the action has not been requested by the game, the execution manager would inform the game of its failure to request the action. This approach would allow game

designers to develop domains that use environment-activated actions without running the risk of waiting for the user for an indeterminate amount of time.

The final limitation that we address here is not one that can be wholly solved outside of the game. In some situations the execution manager must send an action refused message rather than a perform action message. Current implementations of the execution environment simply ignore this message and do not communicate it to the user. In some applications, such as the Intelligent Help System developed by Ramachandran and Young [16], informing the user of the request refusal would be appropriate. In many games, however, informing the user that, “the action you requested cannot be incorporated into the plan at this time,” would be out of context for the established game to player interactions. To lessen this problem, we try to minimize the number of times when an action refused message must be sent. Reactive mediation and proactive mediation are both designed for this purpose. It is possible that new techniques will be developed to make this situation even rarer, but it will always exist to some degree whenever the user is allowed to interact with a plan.

### **5.3. Conclusions**

The Zocalo architecture brings AI planning within reach of games and other applications developed outside of the AI research community. Plan-driven gaming is usually difficult for game developers to achieve but is in large part made simple by Zocalo. If game developers can express the actions of a game world as operators and can describe the desired story as a planning problem, then they can use Zocalo for much of what remains. Zocalo handles most of the work needed to use declarative plans in procedural games, and reduces the remaining requirements down to implementing the execution environment. While we developed this system for use with games, the ability to procedurally execute plans with

small development and resource requirements is useful for a range of other applications as well.

Zocalo's plan execution features are facilitated by the Zocalo Web services, which make it straightforward to integrate AI planning into any application that has a connection to the Internet. Fletcher relieves the client of the burden of providing computational resources for planning and provides an interface that supports interactions ranging from simple to complex. Fletcher is one of the first of its kind, and represents a step towards the larger goal of increasing access to planning technology for application developers outside of the academic research community. This work has been a successful experience in system building and should lead to future extensions that further enhance the combined area of AI and gaming.

## 6. LIST OF REFERENCES

1. D. Box et al., Simple Object Access Protocol (SOAP) 1.1, World Wide Web Consortium (W3C) note, May 2000; <http://www.w3.org/TR/2000/NOTE-SOAP-20000508>.
2. E. Christensen et al., Web Services Description Language (WSDL) 1.1, World Wide Web Consortium (W3C) note, Mar. 2001; <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>.
3. K. Erol, J. Hendler, and D. Nau, "UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning," Proc. 2nd Int'l Conf. AI Planning Systems (AIPS-94), AAAI Press, 1994, pp. 249-255.
4. K. Hammond, "Case-based Planning: Viewing Planning as a Memory Task," Academic Press, Cambridge, MA, 1989.
5. J. Harris, "Proactive Mediation in Plan-Based Narrative Environments," master's thesis, Dept. Computer Science, North Carolina State Univ., 2005.
6. J. Harris and R.M. Young, "Proactive Mediation in Plan-Based Narrative Environments," Proc. Int'l Conf. Intelligent Virtual Agents, 2005.
7. R. Hartanto and J. Hertzberg, "Offering Existing AI Planners as Web Services," GI Workshop Planen und Konfigurieren (PuK 2005), Koblenz, Germany, Sept. 2005.
8. A. Jhala and R.M. Young, "A Discourse Planning Approach for Cinematic Camera Control for Narratives in Virtual Environments," Proc. Nat'l Conf. American Assoc. for AI, 2005.
9. S. Kambhampati, C.A. Knoblock, and Q. Yang, "Planning as Refinement Search: A Unified Framework for Evaluating Design Tradeoffs in Partial-Order Planning," Artificial Intelligence, vol. 76, nos. 1-2, July 1995, pp. 167-238.
10. G. Klyne, J.J. Carroll, and B. McBride, Resource Description Framework (RDF): Concepts and Abstract Syntax, World Wide Web Consortium (W3C) recommendation, Feb. 2004; <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>.
11. D. McDermott, "The Planning Domain Definition Language Manual," tech. report 1165 (CVC Report 98-003), Dept. Computer Science, Yale Univ., 1998.
12. D. McDermott and D. Dou, "Representing Disjunction and Quantifiers in RDF," Proc. 2002 Int'l Semantic Web Conf. (ISWC 2002), Lecture Notes in Computer Science, vol. 2342, Springer, 2002, pp. 250-263.
13. The OASIS UDDI Specifications Technical Committee, "OASIS - Committees - OASIS UDDI Specifications TC," October 2004, <http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm>.

14. J.S. Penberthy and D.S. Weld, "UCPOP: A Sound, Complete, Partial Order Planner for ADL," Proc. 3rd Int'l Conf. Principles of Knowledge Representation and Reasoning (KR'92), Morgan Kaufmann, 1992.
15. M. Pollack and M. Ringuette, "Introducing the Tileworld: Experimentally Evaluating Agent Architectures," Proc. 8th Nat'l Conf. Artificial Intelligence (AAAI-90), AAAI Press, 1990, pp. 183-189.
16. A. Ramachandran and R.M. Young, "Providing intelligent help across applications in dynamic user and environment contexts," Proc. 10th Int'l Conf. Intelligent User Interfaces (IUI 2005), ACM Press, 2005, pp. 269-271.
17. M.O. Riedl, C.J. Saretto, and R.M. Young, "Managing interaction between users and agents in a multiagent storytelling environment," Proc. 2nd Int'l Conf. Autonomous Agents and Multi-Agent Systems, June, 2003.
18. E.D. Sacerdoti, "A Structure for Plans and Behavior," Elsevier/North-Holland, Amsterdam, 1977.
19. A. Tate and J. Dalton, "O-Plan: a Common Lisp Planning Web Service," Proc. Int'l Lisp Conference 2003 (ILC 2003), Univ. of Edinburgh, <http://i-x.info/documents/2003/2003-luc-tate-oplan-web.pdf>.
20. H.S. Thompson et al., XML Schema Part 1: Structures, World Wide Web Consortium (W3C) Recommendation, May 2001; <http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/>.
21. G. Tsoumakas et al., "Web Services for Adaptive Planning," Proc. 16th European Conf. Artificial Intelligence (ECAI'04), Workshop Planning and Scheduling, Frontiers in Artificial Intelligence and Applications, vol. 117, IOS Press, 2005.
22. R.M. Young, M.E. Pollack, and J.D. Moore, "Decomposition and Causality in Partial-Order Planning," Proc. 2nd Int'l Conf. AI Planning Systems (AIPS-94), AAAI Press, 1994, pp. 188-194.
23. R.M. Young et al., "An architecture for integrating plan-based behavior generation with interactive game environments," Journal of Game Development, vol. 1, 2004.

## **APPENDICES**

## 7. APPENDICES

The sections in this chapter contain depictions of the XML schemas used throughout Zocalo. The XML serializations of these schemas are intended for machine processing rather than human reading, so we use tree structures to depict the schema elements and types rather than listing the XML serializations. The schemas currently used in Zocalo are either included in the planning utilities assembly or used in communications between an execution environment and the Execution Manager – Socket Shell; they are organized into two corresponding sections below.

In the depictions found in this chapter, rectangles represent XML elements and rectangles with cut corners represent our own XML element type definitions. The graphic of four dots connected by a horizontal line indicates that the subsequent elements come in sequence, while a vertical line and four dots indicates that the schema requires a choice from one of the elements that follow. The permitted multiplicity of an element, sequence, or choice is indicated below its depiction; if no multiplicity is indicated, then the multiplicity is one.

### 7.1. Planning utilities schemas

The XML schemas that are housed in the planning utilities assembly facilitate the validation of data used in the Zocalo applications. The depictions in the following sections do not represent all of the details in the planning utilities schemas, but they do give the reader an impression of the data included in these documents and how that data is structured. Each schema from the planning utilities assembly is discussed its own section below; in alphabetical order, these schemas validate the documents called *action specifiers*, *domain*,

*environment state, goal state, planning context, plan node, plan space, policy table, predicates, and problem.*

### 7.1.1. Action specifiers

The action specifiers document is used to define a set of actions. By first listing constants and grouping these constants into sets, an action specifiers document can identify actions with specific bindings or actions with parameters that bind to any constant.

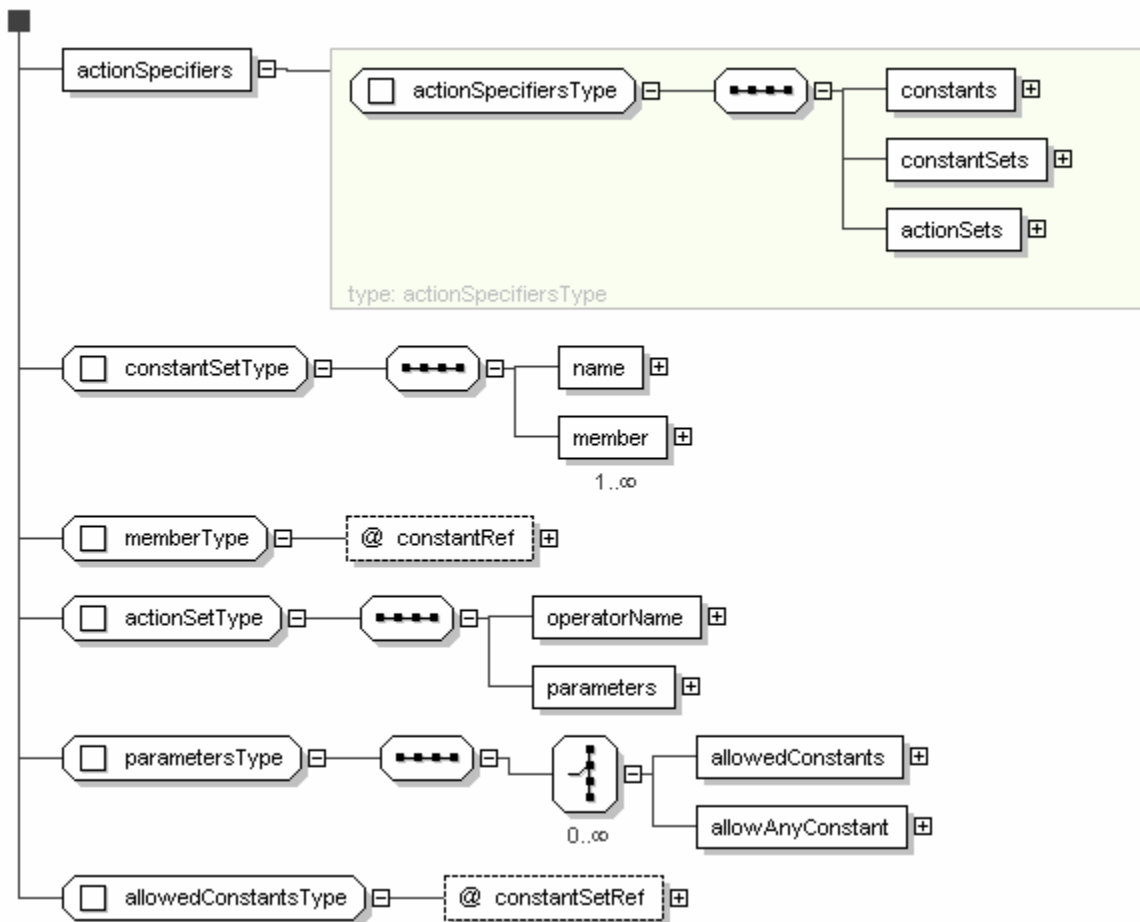


Figure 7.1. The action specifiers document schema.

### 7.1.2. Domain definition

The domain document is used to specify operators for a planning problem. It is based on the input format of the Longbow planner [22].

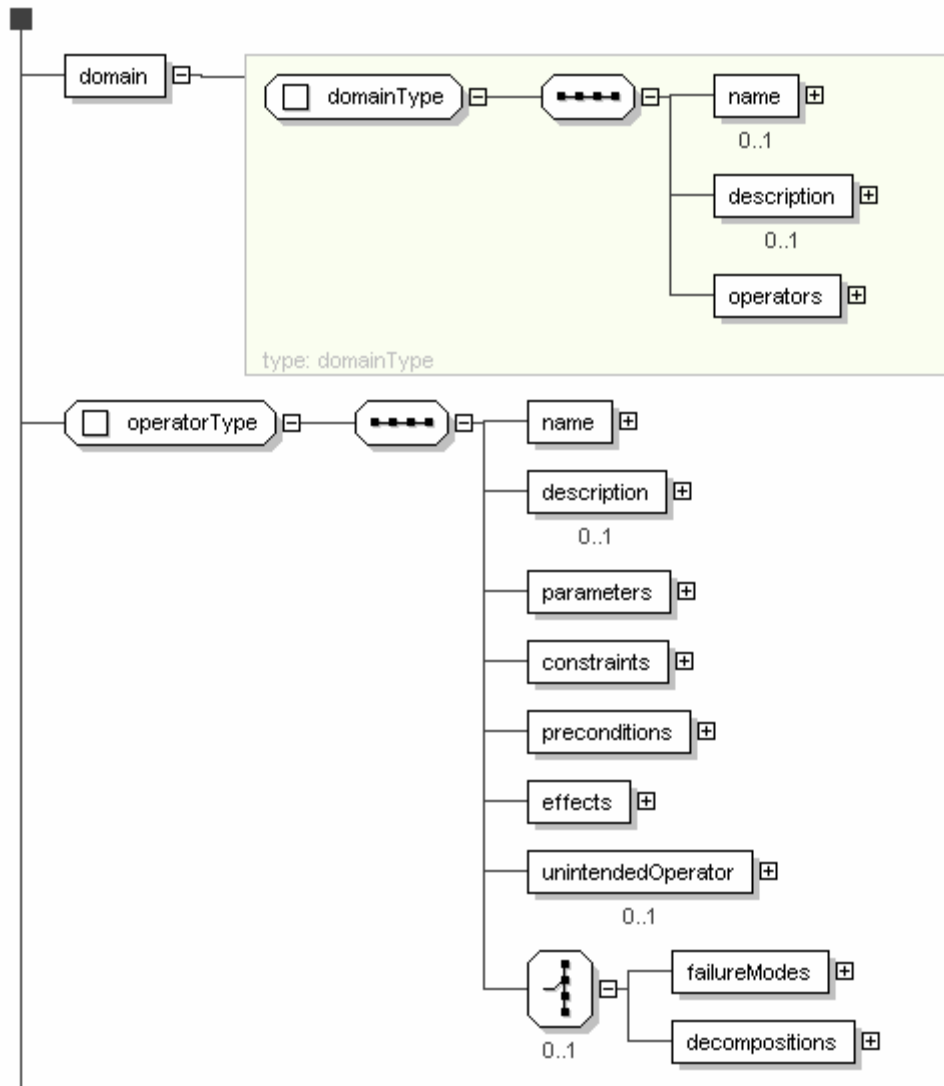


Figure 7.2. The domain document schema, part 1.

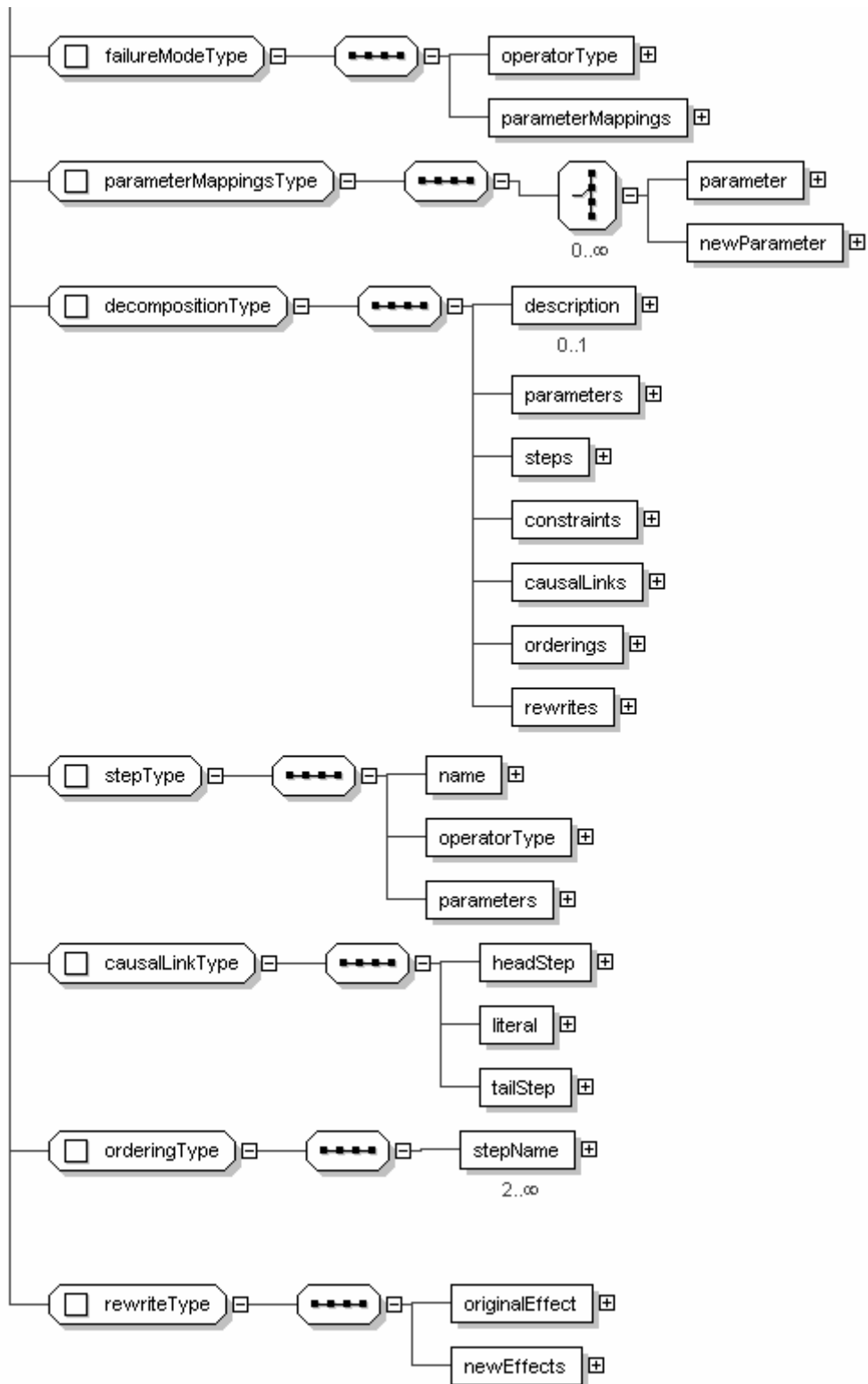


Figure 7.3. The domain document schema, part 2.

### 7.1.3. Environment state

The environment state document is simply a list of positive literals. Applications using this document understand that any literal that is not in this list is false in the state described by the document. This convention is known as the closed world assumption. The XML for a literal is defined in the predicates schema, depicted in Section 7.1.9.

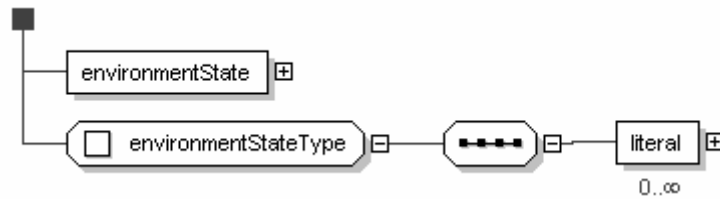


Figure 7.4. The environment state document schema.

### 7.1.4. Goal state

The goal state document is simply a list of literals, each of which may be either positive or negative. This document makes no statement about literals not included in its list. The XML for a literal is defined by the predicates schema, depicted in Section 7.1.9.

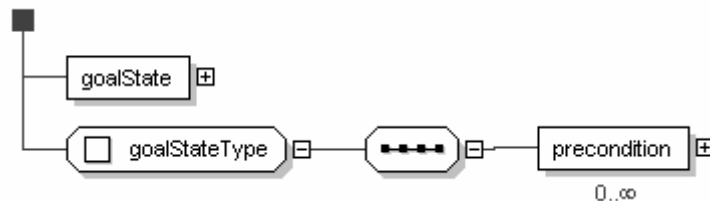


Figure 7.5. The goal state document schema.

### 7.1.5. Planning context specification

The planning context document contains a sequence of elements, most of which refer to documents defined in the other planning utilities schemas. Some of this data may also be

represented in a compressed form using a Base64 encoding. As a third option, planning context identifiers can be used to reference data from existing planning contexts. In the final element, the heuristic search function is specified by including its name in plain text.

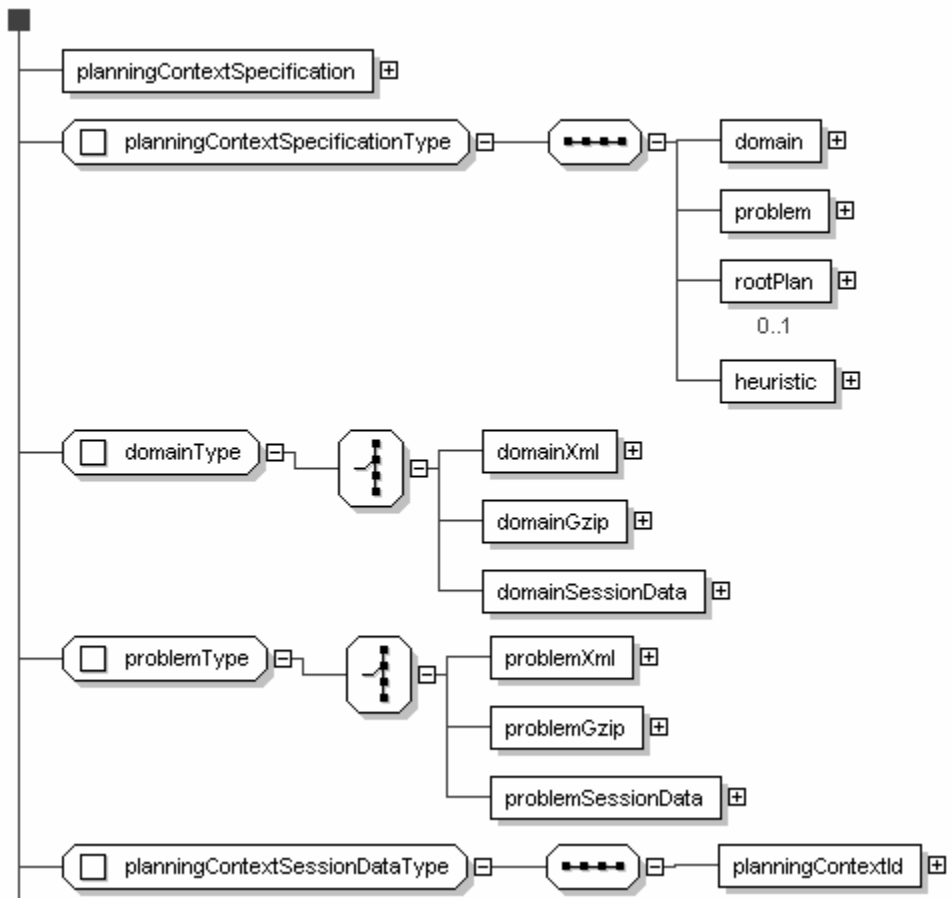


Figure 7.6. The planning context document schema, part 1.

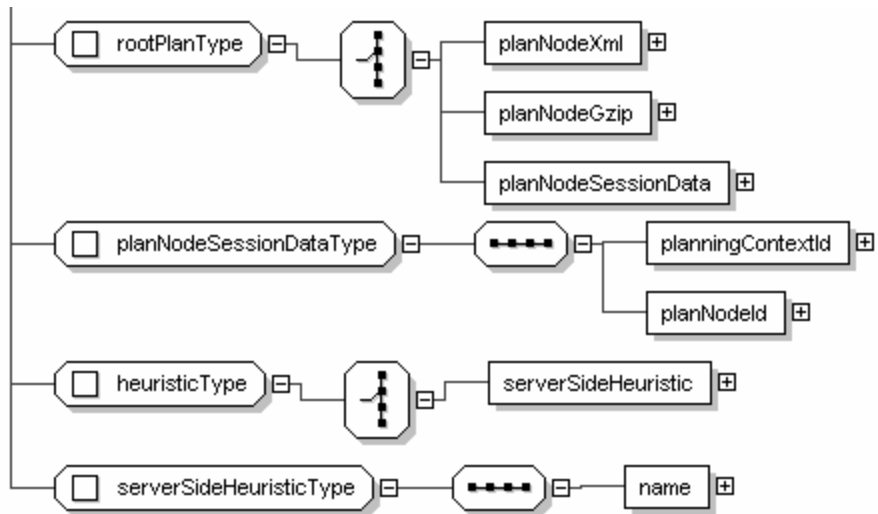


Figure 7.7. The planning context document schema, part 2.

### 7.1.6. Plan node definition

The plan node document contains data concerning steps and the links between those steps. In plans that are not complete, it also contains data identifying the flaws that make it impossible to execute the unmodified plan successfully.

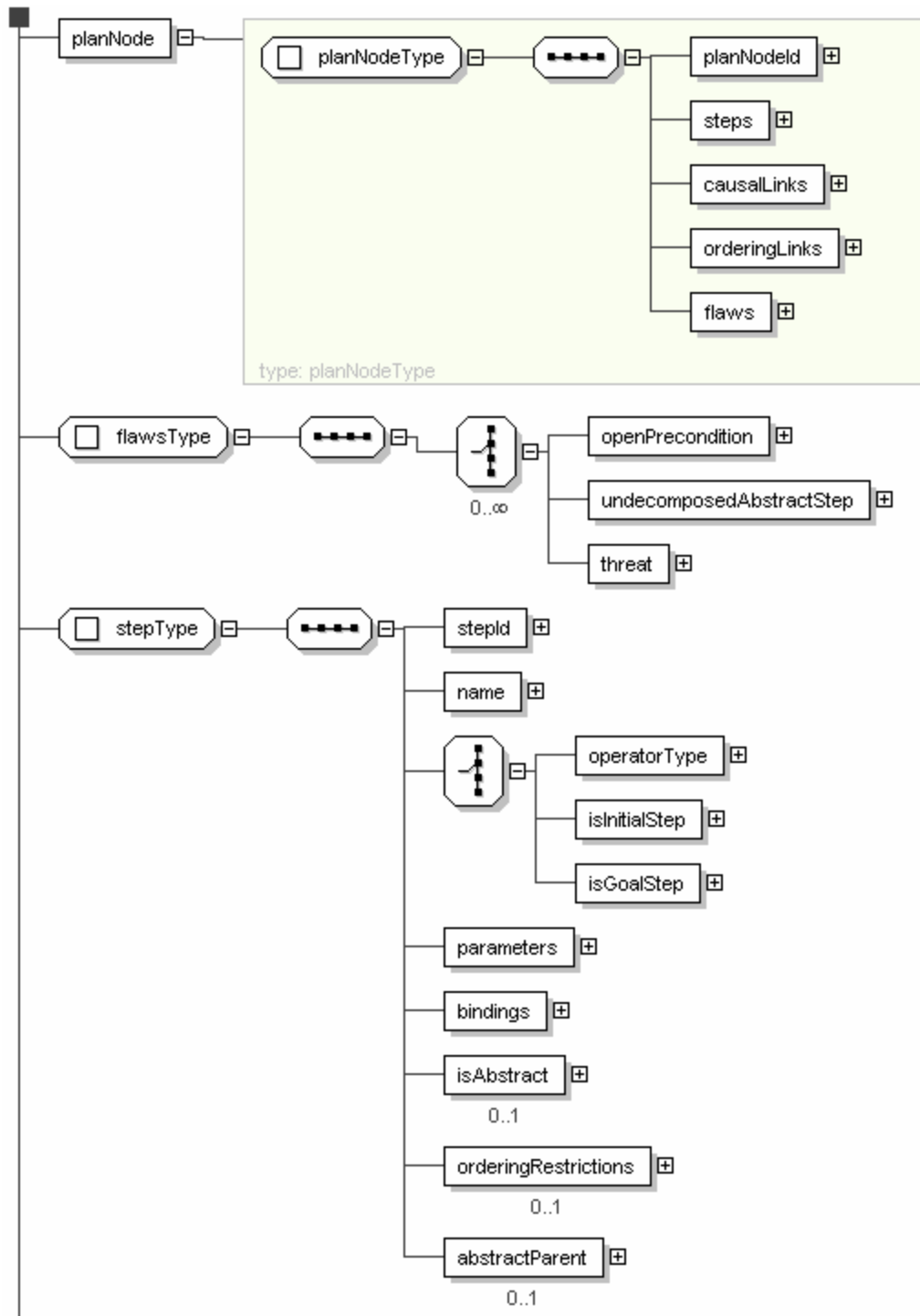


Figure 7.8. The plan node document schema, part 1.

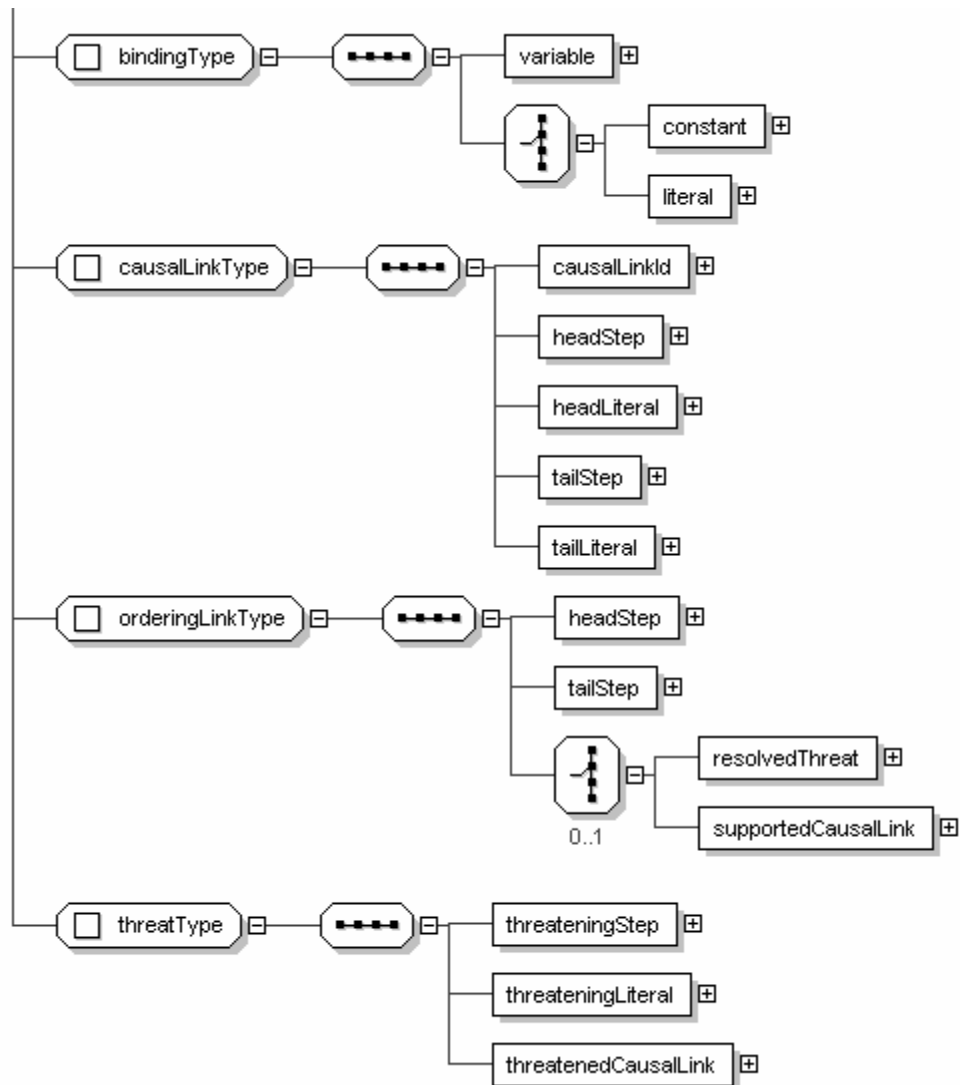


Figure 7.9. The plan node document schema, part 2.

### 7.1.7. Plan space definition

The plan space document is simply a tree structure, with nodes that contain information about plan nodes in a plan space. In addition to the plan node identifier and information on children, this document contains the heuristic value and the number of unresolved flaws for each plan node that it summarizes.

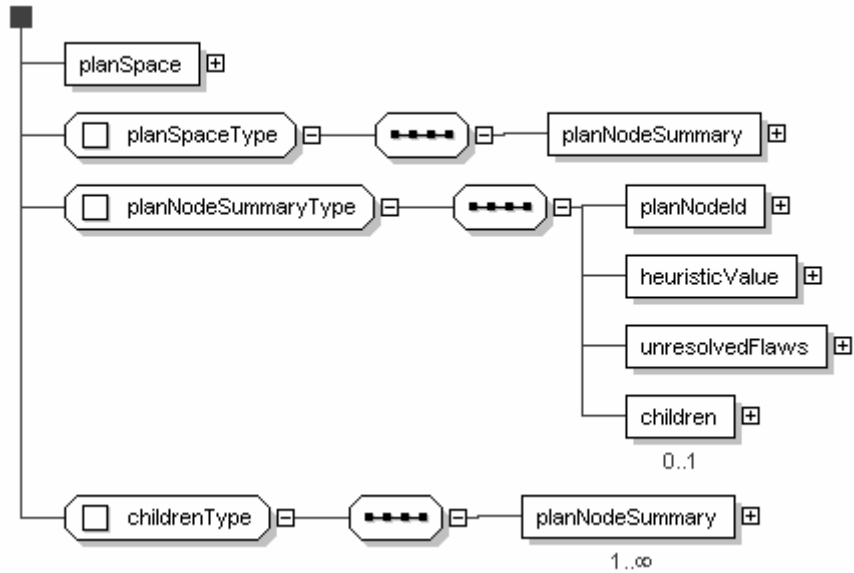


Figure 7.10. The plan space document schema.

### 7.1.8. Policy table

The policy table document contains data for mediation. It lists responses for each exceptional action, for each interval. In addition to the data need to take the response, each response has a list of literals that must hold in order for it to be valid.

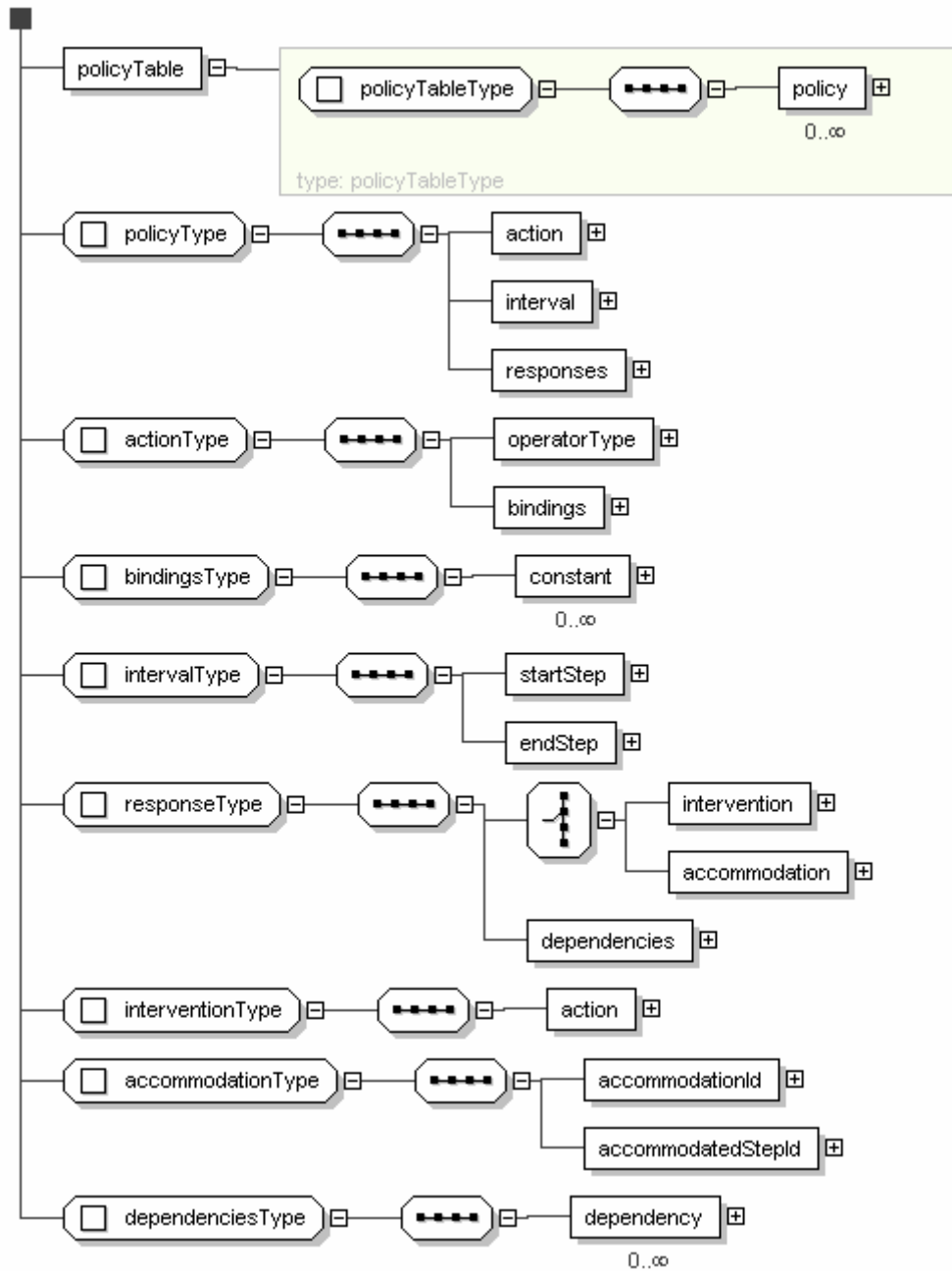


Figure 7.11. The policy table document schema.

### 7.1.9. Predicates

The predicates schema does not define any elements, but it defines types referenced by elements from other schemas. This schema defines the format that we use for literals. A

literal is an atomic sentence or its negation; an atomic sentence is a predicate (a text string) and a sequence of terms. A term may be either a constant (a text string), a variable (a special type of text string), or a literal.

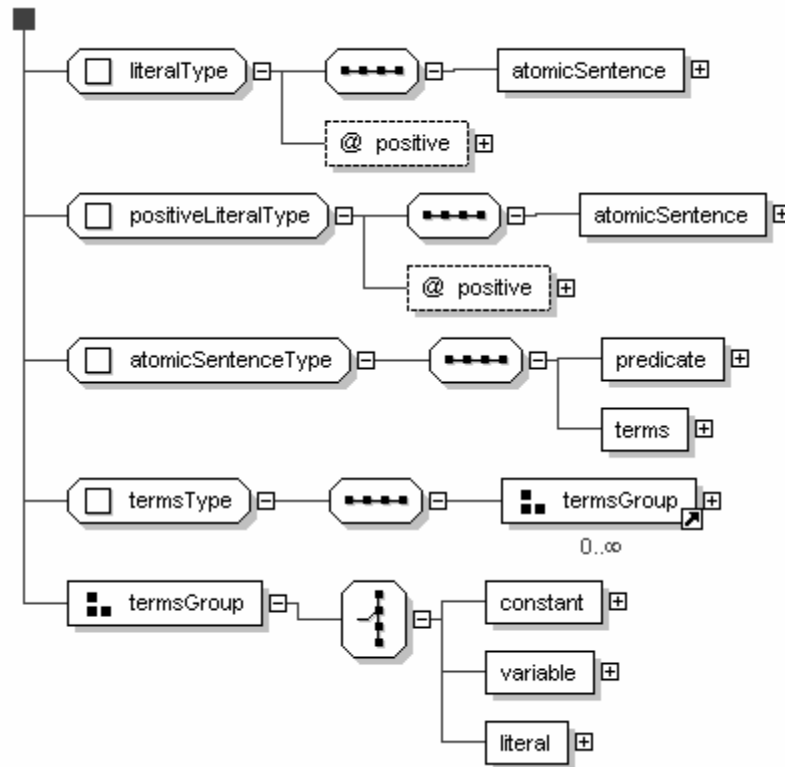


Figure 7.12. The predicates schema.

### 7.1.10. Problem definition

A problem document combines the data found in an environment state document with the data found in a goal state document. It first contains an initial state specification, which is a list of positive literals. The closed world assumption applies to this set of literals, meaning that any literal that is not listed is implicitly false. Second, it contains a goal state specification, which is a list of positive or negative literals. The closed world assumption does not apply to this set of literals.

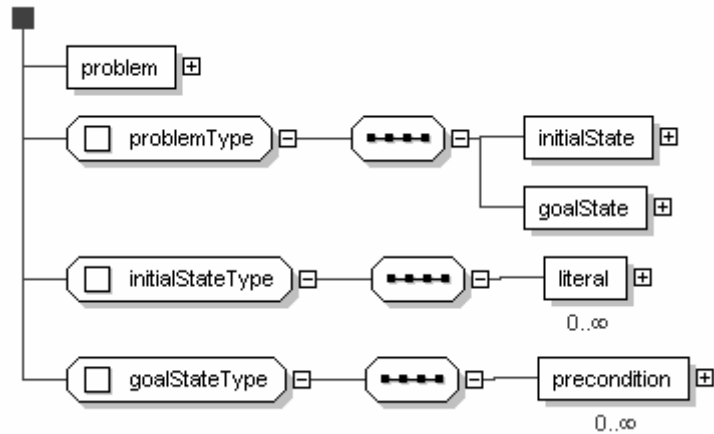


Figure 7.13. The problem document schema.

## 7.2. Execution Manager – Socket Shell XML schemas

The execution environment can choose to communicate with the execution manager via the Execution Manager – Socket Shell application. When these applications communicate, they do so using XML fragments that are defined as elements in the schemas depicted in the following sections. The first section depicts messages received by the execution manager and the second section depicts messages sent from the execution manager to the execution environment. The depictions in these sections do not represent all of the details in the XML fragments used by these applications, but they do give the reader an impression of the data included in these messages and how that data is structured.

### 7.2.1. Received messages

The EM-SS receives input from the game in the form of XML fragments that are defined as elements in the schema depicted below.

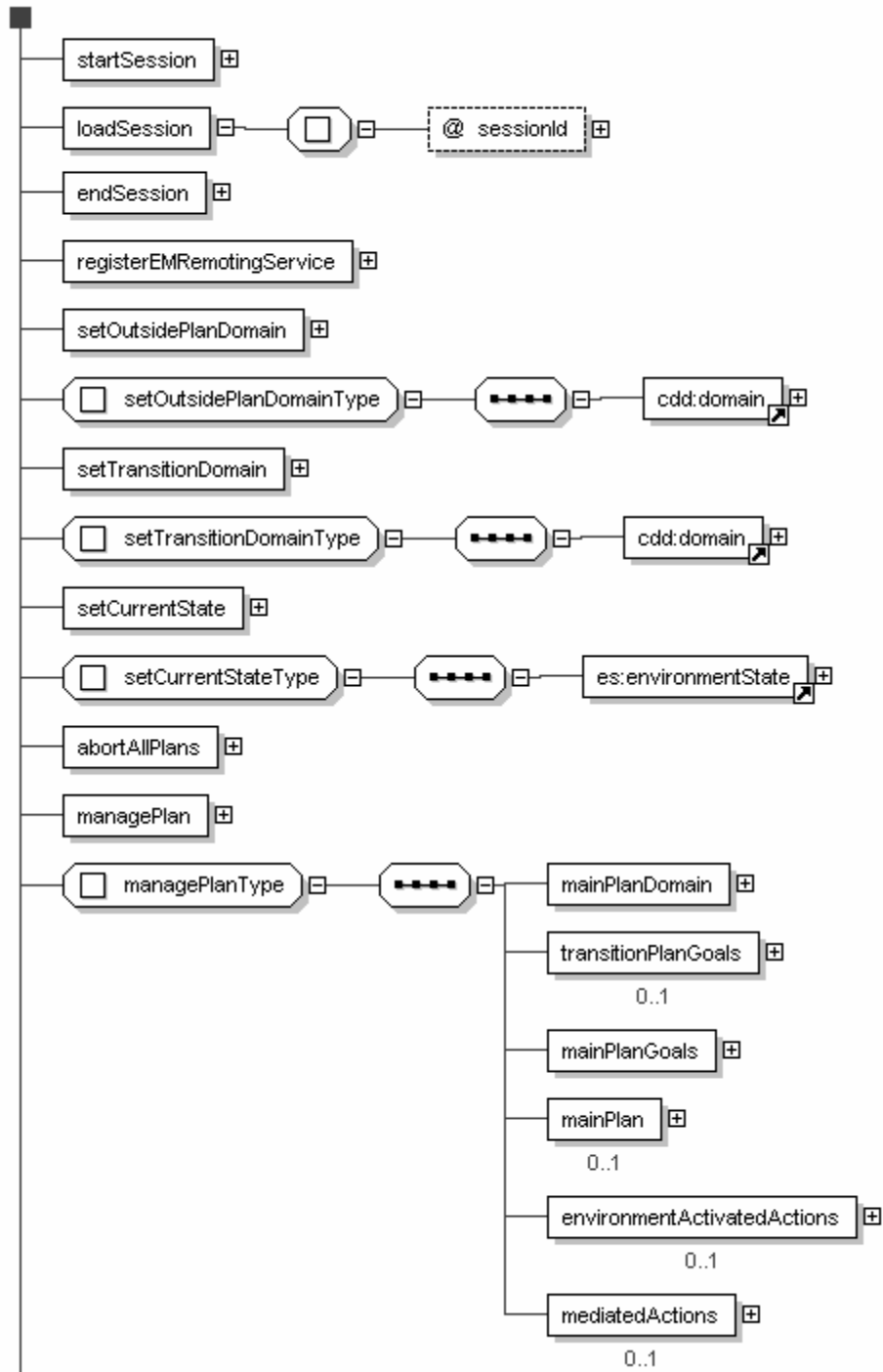


Figure 7.14. Messages received by the EM-SS, part 1.

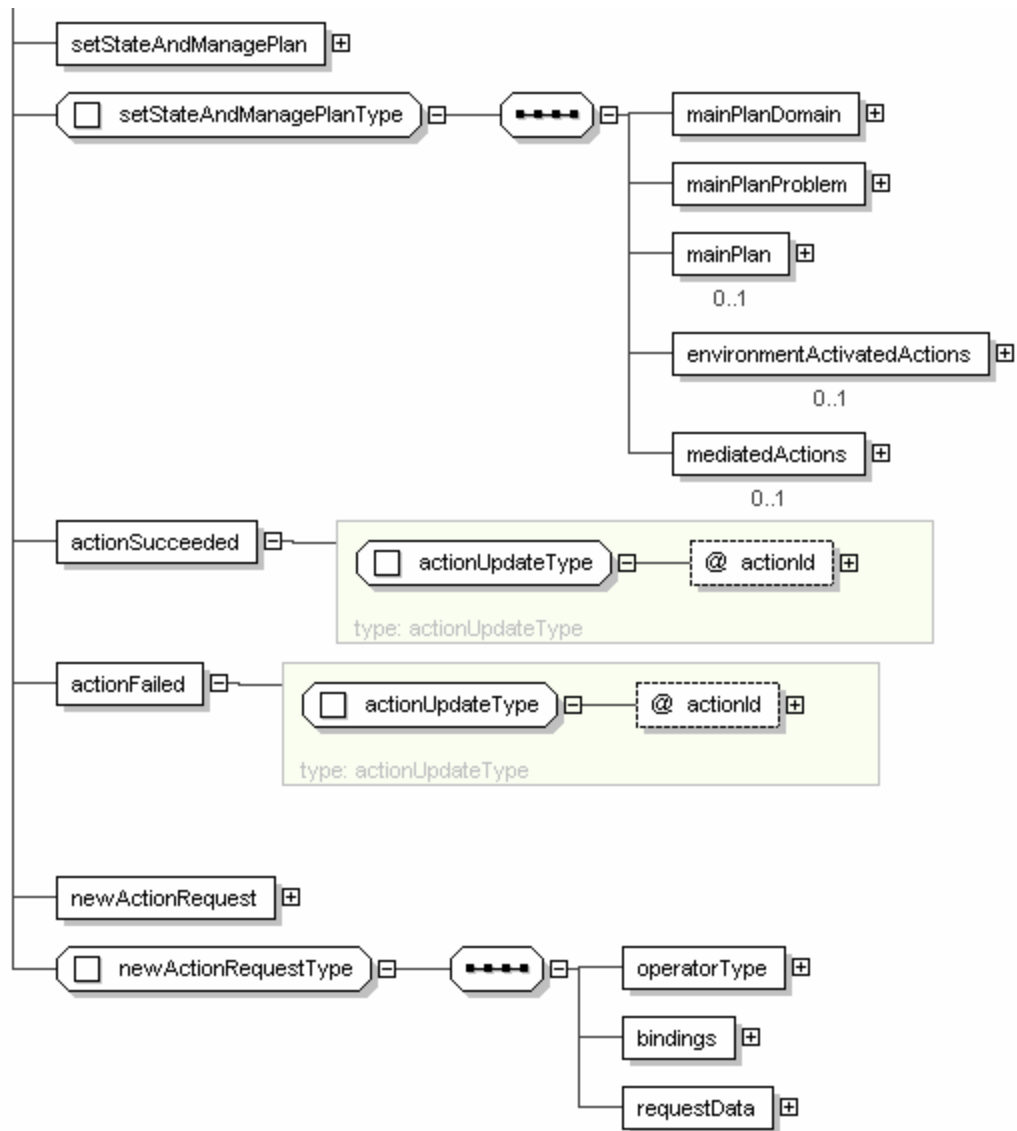


Figure 7.15. Messages received by the EM-SS, part 2.

### 7.2.2. Sent messages

The Execution Manager – Socket Shell sends its directives to the execution environment as XML fragments that are defined as elements in the schema depicted below.

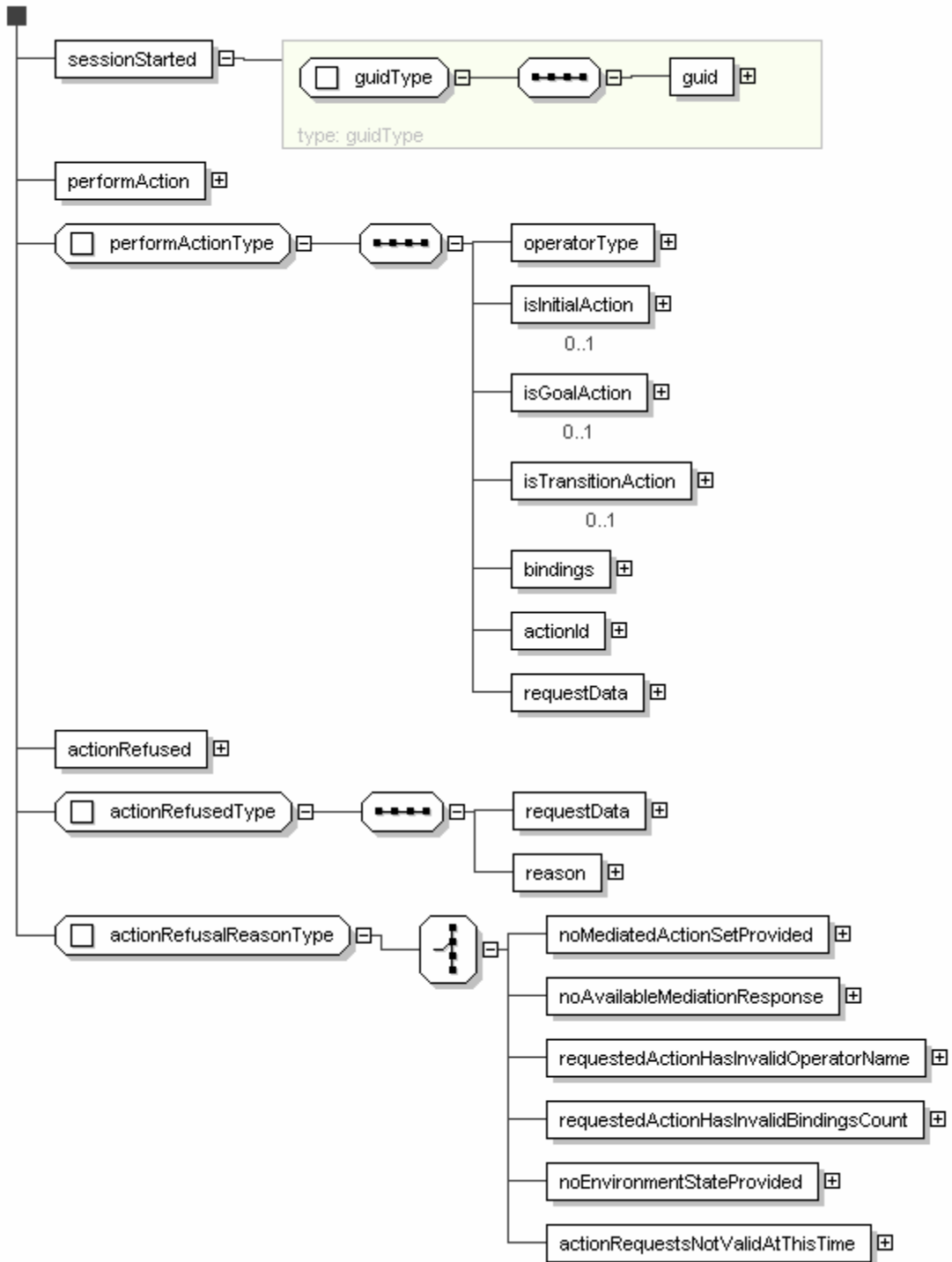


Figure 7.16. Messages sent from the EM-SS.