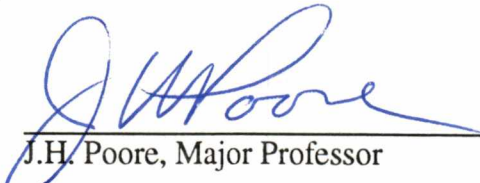


To the Graduate Council:

I am submitting herewith a thesis written by Alex L. Bangs entitled "An Evaluation of Integration of the Trace Assertion Method with the Box Structure Method for Coding in C++." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science.



J.H. Poore, Major Professor

We have read this thesis
and recommend its acceptance:





Accepted for the Council:



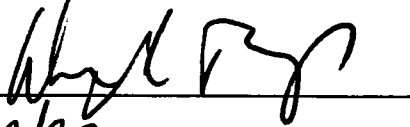
Associate Vice Chancellor
and Dean of The Graduate School

Statement of Permission to Use

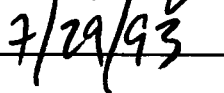
In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature

A handwritten signature in dark ink, appearing to be "W. H. R. G.", written over a horizontal line.

Date

A handwritten date "7/29/93" in dark ink, written over a horizontal line.

AN EVALUATION OF INTEGRATION OF THE TRACE ASSERTION METHOD WITH THE BOX STRUCTURE METHOD FOR CODING IN C++

A Thesis
Presented for the
Master of Science Degree
The University of Tennessee, Knoxville

Alex L. Bangs
August 1993

Dedication

In Memory of Laura M. Bangs, a.k.a. Marg.

Acknowledgments

I owe a great debt to Professor Jesse Poore for allowing me to work with him on software engineering and for getting me interested in the subject in the first place. I want to thank Professor Bradley Vander Zanden for serving on my committee and providing useful comments on object-oriented systems. I also am very thankful to Reinhold Mann and François Pin for allowing me the opportunity to study at the University of Tennessee, being flexible with my work and hours, and giving me the opportunity to pursue research at Oak Ridge National Laboratory. I also thank Professor Roger Brockett for giving me my first research opportunity, and for his encouragement to do graduate studies.

A special thanks is in order for the people in the software engineering seminar who discussed the trace assertion method extensively and helped me understand it well enough to accomplish this work: Stacey Prowell, Ken Sharpe, Hailong Mao, and Steven Jones. I'd also like to thank others at UT that have been helpful throughout my research, including Ethel Wittenberg and Dorsey Bottoms.

My parents have been a continual source of encouragement, especially with respect to finishing my thesis. I appreciate the support they have given me over the years and the emphasis they placed on a good education.

Finally, I'd like to thank Becky, who has been helpful, encouraging, and patient through the many weekend and evening hours I have worked on this project.

This research was sponsored by the Engineering Research Program of the Office of Basic Energy Sciences, U.S. Department of Energy, under Contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

Abstract

The Trace Assertion Method, originated by David Parnas, is a method for developing specifications for software modules. The nature of the method allows verification of consistency and completeness of the specification, and provides a rigid structure to the designer. This method is extended to work with object-oriented designs for a C++ system involving a user interface. A number of object-oriented concepts which are not present in the original Trace Assertion Method are incorporated into the method and demonstrated on two completely specified increments of the system being developed. In addition, the method is incorporated into a system wide view beyond the original modular scope of the method. Advantages of the adapted method and its problems are discussed.

Preface

There were a number of goals that I had set out to accomplish with this work. The overall goals were to learn something about cooperating robots and software engineering, and to construct a simulator in the process of doing both. On the software side specifically, I wanted to experiment with specification methods for designing a real system. I also wanted to experiment with object-oriented design and see how this meshed with the specification methods being explored. Finally, I wanted to work in C++ and X windows, to gain experience with both.

I chose the Trace Assertion Method as the specification method for this work for three reasons. First, I was familiar with it following a seminar at the University of Tennessee in the Spring of 1992. Second, it appeared to be a good candidate for object-oriented specifications, because it grouped functions together in modules in a manner that seemed to mesh well with object-oriented concepts. Third, Professor Poore and I were curious as to how the trace method would apply to real systems.

The work began in the Fall of 1992, with an attempt to build a simulator for cooperating mobile robots. I built a concept prototype of a simulator in January of 1992, but this ran in Smalltalk on a Macintosh and was very slow. In the Summer of 1992, I started to work on a prototype interface using X Windows and C++ under the SunOS environment. This prototype was a good learning experience, and a number of lessons learned from that are included in the requirements specification document for the simulator interface, which was written as the next step (this document is included in Appendix B).

Once the requirements had been established, the hardest work was in trying to adapt the specification method to the software being designed. The first increment is relatively simple—three major objects with some supporting objects. It proved to be more difficult to specify than anticipated, and extensions to the Trace Assertion Method were required to allow true object-oriented designs to be specified. Specifically, notations for handling module I/O and module interactions needed to be more fully developed. In addition, the

method, which was created for module specification, had to be adapted to fit into a full system design.

One of the most helpful ideas came in a brief conversation with Neil Erskine, a student of David Parnas. I was struggling with module interactions, and Neil suggested a new method that I had not considered. While the idea that he gave me required some development to be useful, it nevertheless allowed me to progress significantly.

This thesis documents the work done to adapt the method for object-oriented/C++ design, and gives examples from the increments developed.

Table of Contents

1 Introduction	1
1.1 An Introduction to the Trace Assertion Method	3
1.2 An Introduction to Object-Oriented Design.....	7
1.3 An Introduction to the Box Structure Method	9
1.4 Other Object-Oriented Specification Systems	10
2 Adapting the Trace Assertion Method	11
2.1 Specifying Object Inputs and Outputs	11
2.2 Class Functions	15
2.3 Constructors/Destructors.....	17
2.4 Inheritance.....	18
2.5 Understanding Canonical Traces	19
3 The Specification Process	21
3.1 Background	21
3.2 Design Process Steps	22
3.3 Trace Assertion Method Verification Details	23
3.4 C++ Verification Details	24
4 Case Studies: Classes from WestWorld	25
4.1 PopupMapSize	25
4.2 MapObject Hierarchy.....	35
5 Conclusion.....	48
Bibliography	50
Appendices	53
Appendix A: An Example Specification.....	54
Appendix B: The Requirements Document	58
Appendix C: The First Increment Specification	83
I. Top Level Black Box	84
II. Class Design and Class BB Specifications	87
III. TAM Specifications for Classes	93
IV. Clear Boxes	113
Appendix D: The Second Increment Specification	116
I. Top Level Black Box	117
II. Class Design and Class BB Specifications	122
III. TAM Specifications for Classes	132
IV. Clear Boxes	171
V. Class Design and Class BB Specifications (Second Level).....	172
VI. TAM Specifications for Classes (Second Level)	176
VII. Clear Boxes (Second Level)	192
Vita	196

1 Introduction

This work discusses adaptations to the Trace Assertion Method (TAM) [Parnas89] to handle the specification of C++ programs using object-oriented designs. The examples given are specifications of C++ objects for a user interface being developed under the X Window System, as part of WestWorld, a simulation system for experiments with cooperating robots.

Simulators represent an important part of research in robotics, allowing robot navigation and other tasks to be tested in an environment that is more forgiving of mistakes and allows experiments not possible in the real world. A simulation is typically made up of a number of components that manage both the display for the operator as well as the actual pieces of the robots being simulated. Use of object-oriented systems for designing and implementing simulators is common since the object model allows simulator components and data representations to be more easily and naturally designed.

As a simulation becomes more complex, however, it presents a problem for the designer. A large number of interacting components can be difficult to design and maintain. This is where specification methods are useful. By using specifications for the various components in the system, components can be designed in parallel, without knowledge of the internals of other components, and in the confidence that the interfaces for each component are complete.

The specification method chosen for this development effort was the Trace Assertion Method, as defined by David Parnas and Yabo Wang. Unfortunately, the TAM report leaves many questions unanswered, and does not give very complex examples of the application of the method. Given the method's orientation toward modular design, it was chosen as the method for specifying the object-oriented design of the user interface. In addition, some elements of the box structure design method [Mills88] are utilized as well.

In Chapter 2, adaptations to TAM are discussed. TAM, like many other specification methods, works well for parameterized input and output, but deals poorly with classes that manage user interface tasks, where the input and output are not easily parameterized. In addition, TAM handles input and output as variables, but some forms of input and output may be more in the form of events or functions, rather than variables. This is crucial, since peer interaction between classes through functions is an important part of object-oriented systems. Notational additions to TAM are presented that allow the designer to deal with user interface input and output, and to handle peer interactions between objects. In addition, other object-oriented elements such as class functions, constructors/destructors, and inheritance are discussed and notation for using these elements with TAM is presented.

In Chapter 3, a few example classes from the system being developed are discussed at length, including the adaptations required to allow their specification. TAM was intended for specification of individual modules, not an entire system. A process has been developed to incorporate the TAM-specified classes into a larger, single unit, and to handle inter-class operations. This is discussed in Chapter 4.

After development of a prototype for the interface, a requirements specification was written to document the interface functions and responses. This document is contained in Appendix B.

The first increment developed for the interface involves three classes: Map, PopupMapSize, and WinMap. The Map class contains the core data structure for the application, holding the map dimensions (width, length) and number of pixels per meter (scale) for the map to be represented on the screen. It also controls the X paint window created by WinMap. The PopupMapSize class creates a pop-up window for modifying the Map fields of width, length, and scale. The WinMap class controls the main window for the application, including an X paint window for displaying the map rectangle and menus for controlling the program functions. It acts as the routing point for all menus in the window, and calls functions from other classes based on user actions. This increment was developed in C++ using XView, an X-windows library, and Devguide, an XView graphical user interface (GUI) builder tool. The specification for this increment is contained in Appendix C.

The second increment adds new classes to the first increment to allow the Map to load and save objects in files, and to draw those objects on the screen. Therefore, the Map object was significantly enhanced and objects were added to manage load/save popup windows as well as the data structure for the items in the map. This specification is contained in Appendix D.

1.1 An Introduction to the Trace Assertion Method

The Trace Assertion Method is a system for specification of software modules, as defined in [Parnas89]. It was initially developed by David Parnas [Parnas72] and has been further developed in [Bartussek78; Hoffman88; Hoffman89; McLean84]; note that [Parnas72] and [Bartussek78] also appear in [Gehani86]. [Parnas89] was an attempt to define the method for practical use. In addition, [Erskine92] contains some further refinements.

For the purposes of trace specification, a module is a set of functions designed to work together. Parnas uses the term access programs to refer to the set of functions that can be accessed from outside the module. An event is typically an access program call (there may be other types of events), and an event class is all the possible calls of a particular access program given its set of parameters. A trace of the module is a sequence of input events and their corresponding outputs. For deterministic objects, the trace is usually written with the events only, since the outputs are uniquely determined from an event sequence. A trace can be considered to be similar to a stimulus history for a black box, in that through providing the history of access to the module or box, the outputs of the box can be determined.

A trace can be reduced to contain only those elements which are necessary to produce the required outputs from the module and to maintain the proper sequencing of events. This reduced trace is called the canonical trace, and the reduction is accomplished through equivalences.

An actual trace specification consists of the following parts, some of which may not always be used for all module definitions. First, in the SYNTAX section, the external interface to a module is defined. This consists of tables of input variables, output

variables, and access programs. Second is the CANONICAL TRACES section, which defines the canonical traces for the module. In addition, a DICTIONARY subsection may be included, which defines external types and auxiliary functions. The latter are useful for simplifying trace parsing in the following sections.

The third and typically largest section is the EQUIVALENCES section, consisting of tables of conditions and equivalences for each event class (there is one event class for each access program; there may be multiple events for each class based on parameters to the programs). The form of this section has the left-hand side of the equivalence with a canonical trace T appended with the event class. On the right-hand side is the table of conditions and equivalences. The set of conditions presented in the left-hand column of the table must form a complete partition, and the equivalences in the right-hand column must result in canonical traces, or errors if the event is not allowed under certain conditions. Through the use of these equivalence tables, a trace of the module must be reducible to a canonical trace. Therefore, the equivalences section must correspond with the canonical traces defined.

The fourth section is the VALUES section, with two parts: OUTPUT VALUES and RETURN VALUES. The output values subsection consists of tables of conditions and values. If traces are used in the conditions column, they must be canonical. This means that for unique output values to result from a module, the information required for that output must be present in a canonical trace. This is an important consideration when selecting the canonical traces. The return values subsection details which output values correspond to which access program arguments or return values as defined in the first section.

Once the specification is complete, it must be verified. As Parnas notes, "a principal advantage of this method is that systematic validation of the design is simplified." [Parnas89]. The verification steps are contained in section XVIII of the TAM report, "Assuring completeness and consistency". This section is as follows:

A specification is complete if the values of the output vector are specified for every legal trace. To check for completeness, one must verify that:

- (1) There is one equivalence function for each event class.
- (2) There is one output function that specifies each output value.
- (3) The predicates in the left-hand column of each table partition the intended domain of the relation.
- (4) The predicates in the right-hand column are defined whenever the corresponding predicate in the left-hand column is 'true'.

A specification is consistent if one cannot derive two contradictory statements about the output vector values. Consistency is assured by verifying that:

- (1) The canonical form fulfills the requirements of section XI.
- (2) All traces specified in the right-hand column of the equivalence section are canonical.
- (3) For function definitions, all right-hand sides specify a unique value.

These checks can be carried out systematically and, often, mechanically.

Section XI, mentioned above, notes that "the canonical form must have the property that every legal trace is equivalent ... to exactly one trace in that form." The actual specification verification steps, including canonical trace verification, are detailed in Chapter 4 below.

An Example: The Savings Account Tracker

As an example, consider a simple system to track a savings account balance. This system has three basic functions: DEPOSIT, WITHDRAW, and BALANCE. The DEPOSIT function, given a parameter with the amount to deposit, deposits the money in the account. The WITHDRAW function, given a parameter with the amount to withdraw, withdraws the requested amount or gives an error if not enough money is available. Finally, the BALANCE function reports the current balance, which is the net of the deposits minus the withdrawals. The full specification is contained in Appendix A.

The first step is to specify the syntax for the module. For output variables, there is only one—the balance variable which is returned by the BALANCE function. For access programs, there are three, WITHDRAW, DEPOSIT, and BALANCE.

Second, specify the canonical trace for the system. Since the canonical trace determines the state of the system, it is necessary to consider what functions will need to be remembered in order to achieve the proper functionality. Since the BALANCE function reports the state but does not change it, BALANCE will not be in the canonical trace. WITHDRAW and DEPOSIT both modify the state, so they should be in the trace. Therefore, the canonical trace will be:

$$\text{canonical}(T) \leftrightarrow (T = [\text{DEPOSIT}(x_i)]_{i=0}^n \cdot [\text{WITHDRAW}(y_j)]_{j=0}^m)$$

This can be interpreted as meaning that T is canonical if and only if T is a string of events with zero or more DEPOSIT events followed by zero or more WITHDRAW

events. The "." character is the concatenate operator for traces. Note that the ordering that is imposed in the canonical trace does not necessarily represent the actual order of events arriving at the module. This is addressed below.

As part of the canonical trace section, an auxiliary parse() function can be defined which will simplify the equivalences to follow. The function

$\langle \text{boolean} \rangle \text{ parse}(\langle \text{trace} \rangle T, \langle \text{trace} \rangle D, \langle \text{trace} \rangle W)$

is defined to return true if and only if $T = D.W$ and $D = [\text{DEPOSIT}(x_i)]_{i=0}^n$ and $W = [\text{WITHDRAW}(y_j)]_{j=0}^m$.

Third is the equivalences section. In this section, there must be one equivalence for each input event to the module. The results of the equivalences must be canonical traces or errors. For DEPOSIT, the equivalence is simple. A trace of $T.\text{DEPOSIT}(x)$, where T is canonical, is equivalent to a trace with $\text{DEPOSIT}(x)$ added into the appropriate place in T . For WITHDRAW, the equivalence is more complex—there are two cases, based on whether there is enough money for the withdrawal or not. Given a trace of $T.\text{WITHDRAW}(y)$ and given that the sum of the deposits in T minus the withdrawals in T is greater than or equal to y , this is equivalent to $T.\text{WITHDRAW}(y)$. Otherwise the equivalence results in an error of %insufficient funds%. Finally, the equivalence for BALANCE is the simplest. $T.\text{BALANCE}$ is equivalent to T , since it is not in the canonical trace.

The final section is the values section. For output values, there is one item, the variable balance. Given a canonical trace T , the value of this variable is the sum of the deposits in the trace minus the withdrawals. If the trace is empty, the value is zero. For return values, there is one, which is the value from the BALANCE access program, which is mapped to the balance output variable.

At this point, the specification must be verified. For each table, a set of verification items must be checked. In addition, the canonical trace must be verified against the rest of the specification. These verification questions are documented in the example in Appendix A.

1.2 An Introduction to Object-Oriented Design

There are three basic types of systems based on the object model: object-based, class-based, and object-oriented [Wegner90]. Object-based systems are the simplest of the three systems. The key element of object-based systems is the concept of collecting state data and functions which operate on that data together into a single module. Ada is an object-based language.

In class-based systems the data and function collection forms the definition of a class, which can then have multiple instances or objects, which may differ in their state data. This has great advantages when working with a system that may have many replicated components which have common functions to handle their data—such as elements in a linked list, transactions for a financial database, or objects in a robot world map. A typical class-based programming language will eliminate the need to make special references within object functions to act on object data, thus simplifying the programming task.

Object-oriented systems are defined as those which add inheritance to the object model. Inheritance allows objects with common functionality or data to be grouped together in a hierarchy. For example, consider the objects in a robot world map. These objects may be boxes, cylinders, or complex polygonal shapes. Nevertheless, they all share some common functions, such as `draw()`, which directs the object to draw itself on the screen. In addition, they may share some common variables, such as `locationx` and `locationy`. Therefore, it is best to first define a superclass called `MapObject` which has all the common functions, and then define the subclasses, such as `MapBox` and `MapCylinder`.

The hierarchy can be carried to multiple levels, for example, there could be special versions of `MapBox`. In some cases, specialization is better handled through the object's variables rather than creating a new class; this issue is discussed in numerous object-oriented texts. C++ and Smalltalk are examples of object-oriented languages.

In C++, there is a special concept called an abstract base class, or an abstract superclass. In this kind of class, there cannot be any instantiations because some functions are undefined for that class. For example, in `MapObject`, `load()`, `save()`, and `draw()` are all

undefined for MapObject. Therefore, C++ will give a compile error if there is an attempt in the code to create a MapObject. The purpose of this is to define some functions in this superclass, and then force the subclasses to implement certain functions—if they do not, a compile error will occur.

A few other terms should be presented to complete the description of object-oriented systems. Constructors are used in C++ to initialize objects when they are first created. A constructor is a special function which can be defined by the programmer that is automatically called when the object is created. This allows default variable values, for example, to be set automatically. A destructor is a separate function that also can be defined that is called whenever the object is destroyed. This allows, for example, special memory which was allocated by the object to be deallocated. A similar functionality can be achieved in other object-oriented languages, such as Smalltalk, by specializing the calls that create a new object.

Another important concept is the difference between class and instance functions and variables. In some object-oriented languages, including C++, special functions and variables can be defined at the class level. The functions are available to instances, but do not have any special pointers to instance data as an instance would. Class functions do, however, have access to class level variables. There is only one copy of these variables for the entire class and instances. These variables are typically used for maintaining data such as counting the number of instances which are active.

One feature available in C++ that was not used in this project is overloading. This means having multiple definitions of a function which differ in the arguments they handle. For example, in the MapObject hierarchy, there exists a public variable called next and a function called set_next(). Another way to handle this would be to have a function called next() which, when called with an argument, would set the next variable to that argument. If called without an argument, it would simply return the current value of the next variable. Overloading can in some ways clarify notation, but it can also be confusing. It may introduce a burden on the verification of a system which is undesirable, so it was decided that overloading would deliberately be avoided in C++ specifications.

While the concepts used in object-oriented languages are generally the same, some differences may exist in notation. The notation presented above will be used throughout this thesis. Note that while C++ is the target language for this work, there may be some differences between notation used for C++ and that used here. For example, superclasses and subclasses are called, respectively, base and derived classes in C++. For details on features of particular object languages, see the Appendix in [Booch91].

1.3 An Introduction to the Box Structure Method

The Box Structure Method (BSM) [Mills86; Mills87b; Mills88] was developed by Harlan Mills as part of the Cleanroom development process [Mills87a]. The first step in a box structure specification is a *black box*. The black box describes the possible stimuli to the system, the possible responses, and the transition from stimulus histories to responses. The transitions must use only current stimuli or the stimulus history; no state information can be considered. The stimulus history is then used to develop a *state box*, from which a procedural *clear box* is derived. From this point, blocks are identified in the clear box which should be further refined as new black boxes, and the procedure is repeated with these new boxes.

An important part of BSM is the verification steps. At each step, the current box is verified against the box it was created from. Each state box is verified against the black box, and each clear box is verified against the state box. In addition, there are a few further concepts that are examined during verification and design. First, the boxes should be *referentially transparent*, meaning that each box is independent and does not require knowledge about the design of other boxes. *State migration* is the process of moving state data into lower-level boxes when it is only used in the lower level box. Of course, referential transparency must be maintained. Third, *transaction closure* should be verified to show that the stimuli to a box are necessary and sufficient to produce the required responses. Finally, *common services* should be identified where possible to prevent duplication of code and simplify the design.

In [Hevner93], the BSM model is discussed with respect to object-oriented systems, and object-oriented concepts are compared to box structured concepts. Notational difficulties that are involved in developing object-oriented systems are not directly addressed in the

paper. In particular, only object-based systems, which do not include inheritance, are addressed. In addition, the concept of class versus instance is not addressed.

The method presented here makes two modifications to using BSM to specify a system. First, notational changes are made to the boxes to address object-oriented concepts, and to match better with the trace specifications. Second, the trace specifications are used in place of a state box. The process for combining the two is discussed further in Chapter 4.

1.4 Other Object-Oriented Specification Systems

A number of specification systems have already been developed for object-oriented systems, many of which are extensions to existing specification systems, including Object-Z [Duke91; Rose92], MooZ [Meira90; Meira92], Larch/C++ [Leavens92], and others. Several Z-based object-oriented systems, including [Rose92], are presented in [Stepney92]. A few languages were designed with structures which permit more format specification, the most notable being Eiffel [Meyer90]. Using specifications with Smalltalk is presented in [Cook92]. A good overview of a number of methods as well as general concepts of formal methods for object-oriented systems and an extensive bibliography is presented in [Casais93].

2 Adapting the Trace Assertion Method

The Trace Assertion Method, with its emphasis on specification of a group of access programs in a module and on verification, appears to hold great promise for specifying object-oriented designs, since the object method is also based on grouping related functions together. In practice, however, there are a number of problems. First, it is not clear how to handle the distinction between instance and class functions in TAM. Second, module interactions and user interface input/output require notation that is not defined by TAM. Finally, constructors, destructors, and inheritance require some notational changes.

The specifications for two increments developed using the notational changes discussed below are contained in Appendix C and Appendix D.

2.1 Specifying Object Inputs and Outputs

To understand the original TAM limitations with specifications, it is necessary to examine exactly what are the types of input and output that a module/class/object would have to accommodate, and how they would be handled (or not) under TAM.

The following types of output would be typical for a module: (a) publicly accessible variables, (b) access program return values, (c) user interface/external world output, and (d) calls to outside objects (state modifying). Note that (c) is basically a subset of (d). The following types of input would be typical: (a) publicly accessible variables of other objects, (b) access program parameter values, (c) user interface/external world input, and (d) return values from calls to outside objects.

Examining the TAM document reveals the following two items. First, in [Parnas89], section XIV "The syntax section" (p.8), Parnas defines the following¹:

¹There is no discussion of input tables or event tables, and there are no cases of them in the examples given in Parnas' paper. Input tables and input variables events are, however, covered briefly in [Mills86].

The syntax section consists of an input table, an output table, an access-program table, and an event table.

The input and output tables list the input and output variables and specify their types.

...

The event tables define parameterized classes of events as relations on values of the input variables. Only events in these classes may appear in traces.

Second, in section VI "Communication with objects" (p.3), three modes of communication between the object and the outside world are presented: input variables, output variables, and access programs for sending and receiving information.

Given this information, for outputs, it is apparent that case (a), publicly accessible variables, is handled via output variables and specified via the output table, and case (b), access program return values, is handled by access programs and specified by the access program table and output table. Cases (c) and (d) are more difficult.

For inputs, case (a), publicly accessible variables of other objects, is handled by input variables and specified via the input table, and case (b), access program parameter values, is handled via the access programs and specified via the access program table (and perhaps the input value table, although this is not directly addressed in the TAM report). Again, cases (c) and (d) are more difficult.

In general, directly accessible input and output variables from a module violates normal black box specifications, which normally assume all access to a box is via some function or stimulus. Under TAM, however, input and output variables in class definitions without using access functions, such as public variables in C++, are allowed. This simplifies the specification and design of classes by not requiring an access function for each output variable. It is important, however, that when access functions are not used to access variables, that there be a rigid set of criteria for these variables. These criteria have been defined as follows:

- a variable must be initialized by a constructor or a one-time initialization function;
- following initialization, a variable value must always be defined;
- access to a variable from outside must be read-only, i.e. all state modifications of an object must come from function calls.

The more difficult cases can also be handled, through some adjustments to the TAM notation. First, output case (c), user interface output, can be handled through the use of a special output variable for the interface. External world output, such as writing to devices, can also be handled with special variables. In the Map class, an output variable "(output screen)" was added, and in the output values section, the value for the variable was a description of what the output should look like based on the current trace. In addition, drawings of the expected output can be included.

Case (c) for user input is handled differently. This is a problem because the user input that an object is responsible for (i.e. data input to the fields of a dialog box, which then must be processed by the object that created that dialog box) appears to be an integral part of its state, and therefore must be included in some form as part of the event trace for the object. For example, the class `PopupMapSize` is responsible for a dialog box that has three input fields (width, length, scale) and a Change button. When the Change button is pressed, the data from these fields (the result of user input) must be sent to a different class, `Map`, which returns a value indicating if the values were acceptable or not.

Access functions for `PopupMapSize` objects are needed to initialize the object (i.e. creating the dialog via window manager calls), display the dialog, and take action when the Change button has been pressed. The input fields are not handled directly by the object. Instead, they are handled by the window manager and can be retrieved at any time. Therefore, the reasonable logic is that when the Change button is pressed, the field values are retrieved and passed onto the `Map` object. If the `Map` object accepts the values, they should be redrawn on the screen to represent the new `Map` values.

In this case, however, the user input is not really part of the object's state. It is part of the window manager's state, and it should be considered an input variable to the program which can be retrieved at will. The state maintained by the object is a pointer to the window manager-maintained input field.

In another case, however, the input is again not directly handled by the object, but input directly affects the state of the object. This happens when an `XView Notice` is displayed by an object. There is some sequence of events that leads up to this notice being displayed, and nothing else can occur until this notice is dismissed. Therefore, the

program must wait on the user to press the Confirm button, making this an important input to the system. However, since the notice box and input are handled by XView, the notice output is instead treated as a user interface output from the object, and the confirm input is treated as an input variable event which is part of the trace.

The final cases are input and output cases (d), which involve access calls to external objects which may return values. These calls are the basis of peer interactions between objects and are an integral part of object-oriented designs. In addition, this also includes calls to "common services" where those services maintain some state for the object—i.e. XView, a file manager, etc. Calls to outside objects should be considered as important for specification purposes only if they modify the state or report on the state of some outside object. This excludes, therefore, many external utility calls such as string and math functions.

As an example, consider the `change()` function in `PopupMapSize`. It takes the user input values in the dialog box and passes those along to the Map object, via the `change_size()` function. In addition, it must receive a variable back indicating the success of the `Map::change_size()` function. In effect, the call to `Map::change_size()` is a form of output. In addition, the return value of the function is an important input.

The first solution to this problem that was considered is as follows. First, the parameters passed to Map may be considered as output variables. Therefore, those parameters should be included in the output variable tables. The return values from such a function should be included in the input variables table. To show when an output function call would occur, appropriate sequencing information must be included in the canonical trace to allow the outputs to occur at exactly the correct moment. This complicates the module specification significantly.

The solution chosen instead involves adding new notation to the equivalence tables. The solution, proposed in part by Neil Erskine [Erskine93], was expanded to allow objects to call functions in other objects and receive return values by adding notation to the equivalence section of a trace specification. For example, in the `change()/change_size()` example given above, the equivalence for `change()` should include an `ADD-TO-TRACE( $T_p$ , change_size())`, where `change` has some pointer p to the appropriate Map object. In addition, in the left-hand column (conditions), where the value returned from

change_size must be considered, the function is written there as well. Obviously, if the change_size function is included in the conditions section, a corresponding ADD-TO-TRACE must appear in the right-hand column. This method has also been adapted to class/instance function interactions, discussed below.

2.2 Class Functions

Class functions are functions which do not require an instance to be run, and do not have a set of instance variables associated with them. Instead, these are used to handle overall class operations, such as counting the number of instances or maintaining common variables for all instances. There may be a set of class variables for use by the class or instance functions, but unlike instance variables, there is only one copy of class variables. For XView programming, class functions are required for user interface callbacks, and are therefore an important part of the development of the interface.

When developing a trace specification, the class and instance function specifications should be included in the same module. The reason for this is that class functions and variables are used for tasks directly related to the module and its instances. For example, class variables can be used to store common variables which will be used by the class or instance functions.

For the purposes of XView programming, callbacks must be made to functions in the program to allow the user interface to pass information and actions to the program. Callbacks cannot be made to instance functions. XView is, however, capable of storing a pointer to an instance. When a callback is made, information passed with the parameters allows the instance responsible for the interface component to be determined.

The best way to handle callbacks under C++ is as follows. When a graphical component is being created, a pointer to the instance creating the component is passed to XView. In addition, a class function is passed to XView as the callback function for the component. When the callback occurs, XView passes this pointer back to the callback function. The callback function is then able to call the proper instance function with this pointer. For example, in PopupMapSize, the class function cfChange(), activated by the Change

button on the popup window, gets a pointer to the instance of `PopupMapSize` to be called, and calls the `change()` function for that instance.

To handle class level functionality within a module, the access programs for class functions appear in a separate section called **CLASS ACCESS PROGRAMS**, and any appropriate output variables in **CLASS OUTPUT VARIABLES**. A separate canonical trace is shown for the class functions, if any are defined.

Within the access program equivalences for class or instance functions, if a class function is required to interact with an instance function, or visa-versa, the following notations developed for module interaction apply:

T_i = trace for instance i of a module

T_c = trace for class functions/variables of the module being defined

$ADD\text{-}TO\text{-}TRACE(T_x, function(parameters))$ = add function() to the trace for T_x with the given parameters, where x is determined through some pointer to the object to be called, or where T_x denotes the class function trace for some class (such as the `Utils` class, which only has class functions and no instances).

Note that **ADD-TO-TRACE** does not affect the equivalence for the calling function. **ADD-TO-TRACE** is not, however, guaranteed to call a function in the canonical trace of the called object; this would limit module interactions. Rather, it must call an access function in the object being called, and it is up to the called object to handle the function. This is important since the calling function therefore only needs to know the external interface for the called object, and does not need to know the canonical trace of the called object.

Also note that an instance can modify the T_c trace without having any specific pointer to the trace, since there is no concept of specific instance for a class. This even holds true for functions outside the class calling public class functions. However, for a class function to modify a specific instance trace T_i , it must have some means of determining i .

As with module interactions, if a class-instance interaction requires that a return value from the external function be considered, then these values should be considered in the

left-hand column of the equivalence section where the ADD-TO-TRACE macro is to be used.

One final comment on notation: if class/instance functions were to be translated into Parnas' notation, it would seem that the instance functions are akin to the "named" modules, i.e. have a pointer to a specific instance, while class functions do not have this pointer. This is actually what C++ does, via providing a pointer called "this" which points to the instance and its variables, but the passing of "this" to the instance function is hidden from the programmer. Using Parnas' named notation for all the instance function calls would be cumbersome, since they are far more common than the class functions. Therefore, all functions are assumed to be instance functions unless noted otherwise. In order to distinguish class functions from instance functions, class functions begin with a lower-case "cf", i.e. cfChange().

2.3 Constructors/Destructors

The Map class uses a constructor, which is a function that is run anytime an instance of the class is created. This allows, for example, variables in the instance to be initialized to some value. Therefore, for such a class, the constructor must be added as an access function, and be shown in the canonical trace if necessary. If the constructor is present in the canonical trace, then an empty canonical trace is not possible, since when a new instance is created, the constructor function is run, and therefore the trace will, at minimum, include this function.

Destructors are functions that, when defined, run automatically when an object is destroyed or scoped out of existence. For example, given a class with a destructor, if an instance is created with a 'new' call and then destroyed with a 'delete' call, at the moment that 'delete' is called, the destructor function will be called. When an object is scoped out of existence, such as when an object declared as a local variable is destroyed when the function returns, the destructor will also be called.

Destructors may or may not be part of the canonical trace for a class. This depends on whether they affect the outputs from the class. Typically, it will not affect the canonical trace since after the destructor has been called, the object has been scoped out of

existence and its canonical trace no longer has any meaning. If, however, the destructor results in a change in the user interface output, it should be part of the canonical trace. A destructor function has no arguments or return values. A destructor was used in the Map() class to eliminate objects created over the lifetime of the Map object, such as MapObject subclasses.

Every class responds to a class access program called 'new' which returns a pointer to an instance of the object that is created at run-time. The object can then be deleted via 'delete'. These functions are automatically handled by C++, although they can be redesigned by the programmer if desired. Therefore, new will be specified for a class only when it will be specially designed. Otherwise, it is assumed to be a part of the class interface.

2.4 Inheritance

Inheritance allows classes to be defined as subclasses of some superclass, inheriting functions and variables from the superclass. In specifying the subclass under TAM, a balance must be maintained between repeating information and providing enough information to properly specify the subclass.

In the external interface portion of a trace specification of a subclass, any functions that are inherited must be shown but should also be noted as inherited. If any of these functions are to be overridden, that should also be noted. In the equivalences and outputs section of the trace specification, a function or output should only be included if its behavior will be different in the subclass due to being overridden or interaction with newly defined functions.

One problem with inheritance that can be seen in the MapObject hierarchy is that the trace equivalence specification for a function may change, even though the function is inherited without change from the superclass. For example, the set_next() function has a very simple trace equivalence in the MapObject class since the canonical trace is so simple:

$$T.set_next(n) \equiv set_next(n)$$

In the subclasses, however, it must be redefined to show its interaction with the new canonical trace:

$$T.set_next(n) \equiv set_next(n).L.D \text{ where } parse(T, I, L, D)$$

In this case, we can see that the equivalence must preserve the information in the canonical trace, represented by L and D, which was not present in the canonical trace for the superclass. Despite the difference in equivalences, the function itself does not have to be reimplemented in these subclasses since the actual program code is the same.

2.5 Understanding Canonical Traces

The canonical trace section of a TAM specification contains a predicate that defines which trace sequences are to be considered canonical. The information held in a canonical trace represents the state data required by a module in order to function. The form of the canonical trace is not unlike that of a piece of stimulus history used by a black box.

Where the canonical trace is really used by a module is in the output variables section of a TAM specification. The cases considered for an output value must be based on canonical traces. Therefore, any values reported from a module must be represented in some form in the canonical trace. Consideration of output requirements for modules required for the WestWorld design led to the realization that the state represented in a canonical trace is more complex than just simple variable state.

For example, several of the objects in the system use XView Notify boxes to put a message on the screen until it is dismissed by the user by clicking on a Confirm button. The display of this notice on the screen must be considered an output of the Utils module which creates it. Therefore, it must be represented in the output values section for this module, and something must be present in the canonical trace to show when this notice should appear versus when it should not. In other words, the error condition that led to the notice appearing must be in the canonical trace. In a black box specification, this could be handled by a simple stimulus-response pairing, not requiring any state data. Under TAM, the canonical trace is holding information related to the sequencing of the functions, a more complex form of state than simple variables.

Another case where state information beyond simple variables is required in the canonical trace arises when dealing with functions that return a value. This is considered an output from a module, but it is not necessarily representative of the simple state of the module. For example, there is a `load()` function in the Map class which given a file name, loads objects listed in that file into the Map. If there is a problem with the load process, then the load function must return a FALSE value, otherwise it returns a TRUE value. Since this is an output of the module, it must be calculable from the canonical trace. However, this information is not important to the functioning of the Map module following this function call. Nevertheless, this information remains in the trace, making it more complicated to develop and maintain.

Under TAM, the canonical trace does represent the state of the module, but more than just the values of stored variables. It represents the state of the program, state such as that in a finite state machine. When discovering the canonical trace, it is important to remember that this is true. Unfortunately, this also means that for an object with complex outputs or sequencing requirements, the canonical trace may be quite complex, and may require parsing functions to simplify the equivalence and output sections.

3 The Specification Process

3.1 Background

TAM is a method for specification of individual modules or classes, not entire systems. Therefore, a bridge is required between the specification of the system as a whole and the specification of the classes which will make up the system. Initially, a traditional box structure method process was considered for specifying and designing the top level system, but this has a number of problems. First, it is desirable to avoid specifying the state data for the system until it has been divided into classes. Then, once these modules have been specified without consideration of state, the state discovery process could proceed for each module. Going through a top level state box would require doing state discovery, and then throwing this state out when the individual modules are specified using TAM.

Another problem with going from a top level system to TAM descriptions of modules is that it is hard to go directly into a TAM specification. While the black box approach is not as complete as a TAM description, and lacks the concept of state that is present in the canonical trace, it does allow a simple view of the responsibilities of a class. Therefore, it would be best to perform a black box specification for each class, then a TAM specification, and finally a clear box.

The final step is to connect the top level black box with the individual black boxes for each class. The goal is to take the stimuli from the top level box, and split this stimuli among a set of discovered classes for the system. In addition, class interactions should be considered at this step. This process is in the realm of object-oriented design, and any object-oriented design process will do, as long as it is focused on the responsibilities of each individual object, rather than the state of the objects. An example of such an approach is that proposed in [Wirfs-Brock90]. The output of this step must be black box definitions for each class in the system, including the top level main() program.

In the interface system developed, main() and global variables were grouped into a pseudo-class called Main, and utility functions were grouped into a class called Utils.

Each X window had a class which was responsible for its creation, maintenance, and callbacks. Finally, classes were defined for data representation—the Map and MapObject hierarchy.

3.2 Design Process Steps

The process steps are as follows:

- Create a black box for the entire increment, showing the stimuli to the system and the appropriate responses.
- Verify black box versus the requirements for the increment.
- Identify possible classes/objects in the system (including main()), assign stimuli to these classes, identify inter-class stimuli, and create black box descriptions for the classes.
- Verify that all top-level black box stimuli are assigned to classes and that the lower-level black boxes perform all the top-level operations.
- Verify that all inter-class stimuli used in black boxes match receiving boxes' specifications.
- Create TAM specifications for the classes
 - use black box header to create syntax section
 - create canonical trace(s) using access programs and input events
 - create equivalences based on canonical trace and output requirements; create auxiliary functions and dictionary entries as needed
 - create outputs based on canonical trace
 - verify specification using Parnas Verification Checklist (see below).
- Verify TAM specification to black boxes for each class.
- Write C++ header for each object using TAM specification and verify to specification.
- Create C++ objects and main()
 - write using C++ header and TAM specification
 - verify versus TAM specification and C++ header
 - verify cross-object access is correct.
- Any lower-level C++ classes "discovered" should be developed by creating a black box definition for the class and then designing as described above.

3.3 Trace Assertion Method Verification Details

The method used for verification of trace specifications grew out of seminar discussions over the TAM document [Parnas89]. The rules are divided into completeness and consistency sections.

Completeness

- (1) There is one equivalence for each event class.
[applies to EQUIVALENCES section]
- (2) There is one output function/relation that specifies each output value.
[applies to OUTPUT VALUES section]
- (3) The predicates in the left-hand column (LHC) of each table partition the intended domain of the relation.
[applies to any table with conditions in the specification]
- (4) The predicates in the right-hand column (RHC) are defined whenever the corresponding predicate in the LHC is 'true'.
[applies to any table with conditions in the specification]

Consistency

- (1) The canonical form fulfills the requirements of section XI in [Parnas89], namely that (a) no two traces in the set are equivalent and (b) every legal trace is equivalent to exactly one trace in the set.
[applies to canonical traces]
- (2) All traces specified in the RHC of the equivalence section are canonical.
[applies to tables in the equivalence and output section]
- (3) All RHC values are unique.
[applies to any table with conditions in the specification]

Canonical trace verification, embodied in consistency rule (1), is quite difficult. Proving (a) requires simply showing that no traces in the set are further reducible via the equivalences defined and that none of these traces are exactly equal. Proving (b) is not really possible since it requires looking at every legal trace or type of trace. Instead, it is up to the designer to bear in mind the requirements of (b) when developing the equivalences and canonical trace. In addition, consistency rule (2) constrains the

canonical trace and assists in its design and verification. In practice, the canonical trace is developed based on an idea of what state will be required to gain the proper outputs and sequencing for the system. If an error is made in the canonical trace, it will be discovered when the equivalences and values sections are written. Once these sections are complete, the canonical trace must be re-examined to verify that it is exactly what is required to produce the appropriate equivalences and values.

3.4 C++ Verification Details

When writing the C++ code based on the trace specification, it is important to keep a few rules in mind which are detailed below. In addition, there are a number of sources which detail rules to bear in mind when designing C and C++ code which may also be applicable to the C++ verification process [Henricson92; Koenig89; Trammell93]. The following is a list of items that are specific to this implementation of TAM for C++ and should be incorporated into the existing TAM and C verification methods:

Verification of C++ code versus the TAM specification:

- All C++ class headers match TAM tables
- Access functions implement inputs, outputs, and equivalences properly
- External function accesses match tables.

C++ Coding:

- Proper headers included in file—including project as well as system headers.
- All functions prototyped in class definition or separately in a header.
- All functions used in file match prototype.
- All functions defined in file match prototype.
- All copied/re-used code has variables that are declared.
- All non-obvious code blocks are commented.

4 Case Studies: Classes from WestWorld

In this chapter, the process described above is applied to two class groups, the PopupMapSize class and the MapObject hierarchy. The process steps are listed in *italics*, with the action that was taken for those steps. It is assumed that at this point, the top level black box has already been specified and the separate classes have been identified, as well as their inter-class stimuli. The complete specification for these classes is contained in Appendix D.

4.1 PopupMapSize

Identify possible classes/objects in the system (including main()), assign stimuli to these classes, identify inter-class stimuli, and create black box descriptions for the classes.

This class was identified as necessary to manage the popup window which will allow a user to input changes in the map size and scaling (see Figure 1). The stimuli for PopupMapSize were identified from the top level black box as well as requirements for interactions with other classes in the system. For the top level, this class must respond to the XView callback for the Change button that is part of the popup window. For class interactions, it must display the popup window when called by WinMap, which handles the "Change Map Size" menu callback. In addition, when the Change button is pressed, the user-entered data must be passed to the Map object for acceptance or rejection. Finally, it must have some sort of initialization function which will create the popup so it is ready to be displayed when the "Change Map Size" menu item is selected.

Map Size

Width: <fl%.2f> Length: <fl%.2f>

Scale: __<int>__ ☒ ☒

Change

Figure 1. PopupMapSize Popup Window

The PopupMapSize class is relatively difficult to specify, because of its user interface interaction and its interaction with another class. In the popup window it manages, the user can type in fields, dismiss the window, or press the Change button to register the changes made. Only the latter action, pressing the Change button, causes the program to act. All other handling, such as the dismissing or basic keyboard events, are handled internally by XView. When a change is made, it has to be handed off to another class, Map, that handles the actual values. Therefore, it uses a pointer to an instance of this class, and when the Change button is pressed, the entered values are passed along. In addition, the Change button cannot call the instance that created the window directly. Instead, it must call a class function which in turn will call the appropriate instance function.

The next step is to define a black box for this class, with a header structure that is similar to the header for the trace specification. The sections of the header include: access programs, output variables, output, class access programs, class output variables, class output, input variables, and external access programs. The construction of a black box, and especially its header, is typically an iterative process, with some items not being included until the transitions are being written and one can see that, for example, certain external inputs or program accesses are required for the class. In addition, some of the specific information required in program arguments, such as the format of the cfChange() callback, requires some knowledge about XView programming. Such knowledge is best obtained through implementing a previous system or through performing experiments to better understand the package that will be used.

This is the header for the black box PopupMapSize:

```

access programs
    void init(Xv_opaque owner_frame, Map* pMap)
    void show()
    void change(Panel_item)

output
    popup window

class access programs
    static void cfChange(Panel_item, Event)

input variables
    Attr_attribute Main::INSTANCE;
```

xv_get variables FRAME_CMD_PUSHPIN_IN, XV_KEY_DATA, entered_width,
entered_length, entered_scale

external access

```
int Map::change_size(Xv_opaque frame, double new_width,
                    double new_length, int new_scale)
double Map::width
double Map::length
double Map::scale
```

Note that any functions are specified using C++ notation, and that all data types are C++ types, although some may be defined by this program or by XView. For example, Xv_opaque is a special type for XView which may contain any of a number of different types of pointers to XView data. Another C++ notation uses the :: operator, as in Map::width; this expression references the width member of the Map object.

After the header comes the transition section of the black box, which specifies how the class responds, given a stimulus history and a current stimulus. The transition for this black box is as follows:

$S_i = \text{init}(o, p) \rightarrow$ no response.

$S_i = \text{show}() \rightarrow$

display popup screen with owner o, with values in width/length/scale fields from p->width, p->length, p->scale, where $(\exists S_j \mid (j < i) \wedge (S_j = \text{init}(o, p)) \wedge \text{not}(\exists S_k \mid (j < k < i) \wedge (S_k = \text{init}(o, p))))$

$S_i = \text{change}(\text{item}) \rightarrow$

given pointer to popup input fields for width/length/scale and popup frame "f" created by init $(\exists S_j \mid (j < i) \wedge (S_j = \text{init}(o, p)))$, call p->change_size(f, entered width, entered length, entered scale); if change_size returns 1 and xv_get parameter FRAME_CMD_PUSHPIN_IN from f is 1, then call show(); if change_size() returns 0, send an error to XView via item to hold the popup on the screen.

$S_i = \text{cfChange}(\text{item}, \text{ev}) \rightarrow$

call PopupMapSize* p->change(item) where p = xv_get(item, XV_KEY_DATA, INSTANCE) [xv_get() is an XView function to get the value of a variable maintained by XView]

Verify that all top-level black box stimuli are assigned to classes and that the lower-level black boxes perform all the top-level operations.

The only top-level stimulus assigned to this box is the Change button callback, which was implemented in `cfChange()`. In addition, the box displays the `PopupMapSize` dialog, which is required in the top-level box.

Verify that all inter-class stimuli used in black boxes match receiving boxes' specifications.

This requires checking other classes to ensure they reference the class being specified correctly, and that any external accesses from this class are done correctly. For external accesses, the `change_size` call to `Map` as well as the direct variable accesses must be verified to ensure that types match.

Create TAM specifications for the classes
- use black box header to create syntax section

This is perhaps the most time consuming step of the specification process. The first step is to take the black box header above and create the syntax section of the trace specification. This should be a direct mapping. In the process of developing a trace specification, it is possible that some extra items might be needed in this section that would not normally be part of the black box.

The tables for the syntax section are shown below. Note that some additional information is added beyond that included in the black box. Specifically, an access column has been added to input variable and output items to indicate how the variable is accessed by this class or may be accessed by other classes. In addition, the `entered_width/entered_length/entered_scale` variables, which were mentioned in the black box but not really carefully defined are enumerated here. They probably should be added into the black box header; it is up to the designer to make these backward compatible steps. The rigor is more important at this step than in the black box.

Finally, a new variable `change_error` has been added. This variable is equivalent to the result from the `Map::change_size()` function. This variable has an important effect on the output from the class, so it has been specified specially as a separate item so it can be included in the canonical trace.

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
init	<void>	<Xv_opaque> owner_frame	<Map*> pMap
show	<void>		
change	<void>	<Panel_item> item	

OUTPUT

Variable Name	Type	Access
(popup window)	(XView Popup window)	N/A

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
cfChange	<void>	<Panel_item>	<Event>

INPUT VARIABLES

Variable Name	Type	Access
change_error	<int>	input pseudo-event
Map::width	<double>	direct access
Map::length	<double>	direct access
Map::scale	<int>	direct access
entered_width	<char *>	XView xv_get value
entered_length	<char *>	XView xv_get value
entered_scale	<int>	XView xv_get value
FRAME_CMD_PUSHPIN_IN	<int>	XView xv_get value
XV_KEY_DATA	<Xv_opaque>	XView xv_get value
INSTANCE	<Attr_attribute>	direct access

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4
Map:: change_size	<int> change_error	<Xv_opaque> popup_frame	<double> new_width	<double> new_length	<int> new_scale

- create canonical trace using access programs and input events

The canonical trace is built by examining the stimuli to the class and determining what events will be needed to produce the required outputs or meet any sequencing requirements for the stimuli. Examining the use of stimulus history in the black box may be useful for this step. For items that have both class functions as well as instance functions, there are two canonical traces, one for each.

The instance canonical trace for PopupMapSize is:

$$\text{canonical}(T_i) \leftrightarrow (T_i = _) \vee (T_i = \text{init}(o,p)) \vee (T_i = \text{init}(o,p).\text{show}()) \vee (T_i = \text{init}(o,p).\text{show}().\text{change}(it)) \\ \vee \\ (T_i = \text{init}(o,p).\text{show}().\text{change}(it).\text{change_error})$$

This canonical trace has five basic forms representing five basic states for the class. It can be empty; initialized; initialized and displayed; initialized, displayed, and have a valid change; and initialized, displayed, and have an invalid change. The `change_error` variable is a function result rather than a direct stimulus to the class, but it must be included here to ensure that the output is correct.

The `show()` function is included in the trace since it must be called before a `change()` call can be made. While a `change()` call before `show()` would not have any ill program effects as long as `init()` had been called, it is not really possible for this to occur since it can only be called after the Change button has been pressed, and this button will not appear until `show()` has been called at least once. The `change()` function will not be called if the user dismisses the popup (something beyond the control of the program) without `show()` being called again.

The class canonical trace is:

$$\text{canonical}(T_c) \leftrightarrow (T_c = _)$$

The canonical trace is empty, which indicates that the class-level operations will not require any state data.

- create equivalences based on canonical trace and output requirements; create auxiliary functions and dictionary entries as needed

In the equivalences section, there is an equivalence for each access program and input event. The equivalences are responsible for resolving any trace into a canonical trace, and therefore are largely verified against the canonical trace. Since the canonical trace is somewhat complicated, it can be more easily referenced in a parsed form, giving rise to the need for a parse function to be defined. This is defined as an auxiliary function as follows:

parse(S,S1,S2,S3,S4) =

conditions	equivalences
(S = S1.S2.S3.S4) $\wedge$ (S1 = [init(o,p)] _{i=0} ¹) $\wedge$ (S2 = [show()] _{i=0} ¹) $\wedge$ (S3 = [change(it)] _{i=0} ¹) $\wedge$ (S4 = [change_error] _{i=0} ¹)	true
else	false

The following tables are the equivalences for this class:

T.init(o,p) $\equiv$

conditions	equivalences
T = _	init(o,p)
T $\neq$ _	%already_initialized%

T.show() $\equiv$

conditions	equivalences
T = _	%uninitialized%
else	I.show() where parse(T, I, S, C, CE)

T.change(it) $\equiv$

conditions	equivalences
T = _	%uninitialized%
T = init(o,p)	%undisplayed%
parse(T, I, S, C, CE) $\wedge$ S $\neq$ _ $\wedge$ I=init(o,p) $\wedge$ p->change_size() = TRUE	equivalence = I.S.change(it); ADD-TO-TRACE(T _p , change_size(f, atof(entered_width), atof(entered_length), entered_scale)) where f is frame created by init()
else	equivalence = I.S.change(it).change_error; ADD-TO-TRACE(T _p , change_size(f, atof(entered_width), atof(entered_length), entered_scale)) where parse(T, I, S, C, CE) $\wedge$ I=init(o,p) $\wedge$ change_error = change_size() $\wedge$ f is frame created by init()

T.change_error $\equiv$

conditions	equivalences
T = init(o,p).show().change(it)	T.change_error
else	%undefined%

T_C.cfChange(item,e) $\equiv$ T_C; ADD-TO-TRACE(T_p, change(item))
where PopMapSize* p = xv_get(item, XV_KEY_DATA, INSTANCE);

- create outputs based on canonical trace

For each item listed as an output or output variable for the class, there must be an output table. Since the conditions side of an output table may only use canonical traces, the

information required to derive an output must be contained in the canonical trace. At this stage, the process is checking that the canonical trace has this information.

The values section has two parts. The first part contains a table for each item in the outputs. The second part maps those outputs to access function parameters or return values. For this object, since no access function returns a value, there are no return values listed.

V[popup_frame](T) =

conditions	values
T = _	%undefined%
else	frame created via init function

V[(popup_window)](T) =

conditions	values
T = _	%undefined%
T = init(o,p)	%undisplayed%
T = init(o,p).show()	popup window displayed on screen; Width field = p->width formatted "%.2f"; Length field = p->length formatted "%.2f"; Scale field = p->scale; values may be modified by user
T = init(o,p).show().change(it) ^ xv_get(frame created by init(), FRAME_CMD_PUSHPIN_IN) = TRUE	popup fields set to values from p-> as given above
T = T1.change(it) ^ xv_get(frame created by init(), FRAME_CMD_PUSHPIN_IN) = FALSE	popup window disappears from screen
else	popup forced to remain on screen, with values as modified by user

In the table for V[(popup_window)](T), the else case represents the case where the change_error function is in the canonical trace in order to indicate when an error has been returned from Map::change_size() and therefore when the popup should be left on the screen until corrected by the user or dismissed.

- verify specification using Parnas Verification Checklist.

The verification for the trace specification is done in two ways. First, the canonical trace is verified through the creation of the equivalences and outputs sections, which will show if the canonical trace is insufficient to provide the required information for output and sequencing. Unfortunately, there is not really a good method to verify whether a

canonical trace contains too much information, except through careful examination and use in the specification.

Second, for each table used in the specification, a series of questions regarding completeness and consistency must be answered to ensure that the table is correct. These questions and answers, mentioned in Chapter 3, are written in the specification directly following each table. See Appendix D for the verification of PopupMapSize.

Verify TAM specification to black boxes for each class.

First, the trace syntax section should be compared to the header for the black box to verify that all information is the same. Second, the output section of the trace specification should be verified against the responses from the black box to ensure that they are the same and are given under the same conditions.

Write C++ header for each object using TAM specification and verify to specification.

The C++ declaration of the PopupMapSize class looks like (note that // is a comment marker in C++):

```
class PopupMapSize {
    Xv_opaque    frame;                // holds XView pointer to main structure for popup
    Xv_opaque    controls;             // holds XView pointer to controls area on popup
    Xv_opaque    map_width_field;      // holds XView pointer to field for entered width
    Xv_opaque    map_length_field;     // holds XView pointer to field for entered length
    Xv_opaque    map_scale_field;      // holds XView pointer to field for entered scale
    Xv_opaque    change_button;        // holds XView pointer to Change button

    Map*    pMap;                      // holds pointer to Map object
    void    update();                  // update numbers in the window (private, for internal use only)

public:
    void    init(Xv_opaque owner, Map* pTheMap);
    void    show();                    // redisplay the box, and do an update
    void    change(Panel_item);        // change button pressed; send values to pMap

// class functions
    static void cfChange(Panel_item item, Event *event);
    // XView button callback for Change
};
```

The first part of the class declaration defines the items private to the class, i.e. variables and functions that are not available to functions outside the class. The section following `public:` defines what is available to outside functions. The private variables are discovered in the process of developing the C++ code. The public interface can be verified easily against the trace syntax given above.

Create C++ objects and main()

- write using C++ header and TAM specification

The C++ code is not contained in this document, but it is available on request. The functions perform the following tasks:

- `init()` creates the window but does not show it on the screen (should only happen once in lifetime of instance).
- `show()` displays the window on the screen and displays the current values from the Map object (`Map::width`, `Map::length`, `Map::scale`) in the input fields; `show()` is called via a menu item which causes the window to "pop up"
- `change()` takes the values of input fields, converts them to numbers, and calls the `Map::change_size()` function with the new values. If the change is successful and the window is still displayed, the fields on the screen are updated to show the values changed to, via `show()`. This is required to get rid of any spurious non-numeric characters that might be entered but ignored by the numerical conversion routines. If the change is not successful, the popup window is forced to remain on the screen.
- `update()` is a private function and therefore only available to functions for this class. It updates the fields on the popup from the Map object values.
- `cfChange()` is a class function which looks up the instance that the button pressed belongs to, and calls `change()` for that instance. The only output from this function is via `change()`, but it does not return a value; rather, it calls another function and via this method passes the information to the other object. This output is dependent on user input, which is not shown in the class definition.

- verify versus TAM specification and C++ header

- verify cross-object access is correct.

Verification of the C++ code against the specification takes two forms. First, the code must be examined to ensure that it has been written to implement the class as defined by

its own header, i.e. check the syntax of the functions and type of the variables that are contained in the class definition. Second, the code must implement the TAM specification, in that sequencing and output conditions are met by the C++ code. Finally, any external access made by the class being developed must be checked to ensure that calls match external functions or classes accessed.

Any lower-level C++ classes "discovered" should be developed by creating a black box definition for the class and then designing as described above.

None were discovered for this class.

4.2 MapObject Hierarchy

(From the Map class) *Any lower-level C++ classes "discovered" should be developed by creating a black box definition for the class and then designing as described above.*

The interface requires the Map class to store information on the individual items in the Map. These items may be of varying types, but they will have some common features. For example, they will all have to have a draw() call through which they will draw themselves on the screen. A case like this is best managed through inheritance, where a single root class is defined that has common functions that will be the same for the subclasses defined, and has stubs for functions that must be defined by the superclass. In the second increment, there are only two types of items that can appear in the map—boxes and cylinders. Therefore, the class hierarchy looks like:

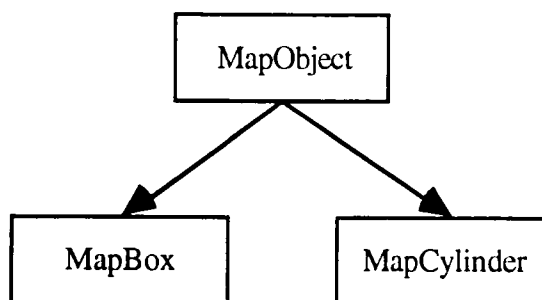


Figure 2: MapObject Hierarchy

The functions and data that will be necessary for both classes must be identified and placed in MapObject. The functions that will be exactly the same for both should also be fully defined by MapObject. Functions that are not defined in MapObject should have some error associated with their response in MapObject.

Each object will require a draw function to draw itself on the screen, a load function to set its internal values according to a string from a file, a save function to return such a string for saving to a file, and a mechanism for allowing the objects to be placed in a linked list. In addition to these instance functions, a class function is required for each subclass to examine a string and indicate whether it is possibly a string that defines an object of that type. At the MapObject level, a class function is also needed to handle the process of selecting a subclass to define a new object being loaded.

The black box header for MapObject is as follows:

```

access programs
    MapObject()
    MapObject* set_next(MapObject* nextobj)
    virtual int load(Xv_opaque frame, int lineno, char* line)
    virtual void save(char *buffer, int bufsize)
    virtual void draw(Display *display, Window xid, int scale)

output variables
    MapObject* next

class access programs
    static MapObject* cfSelectAndLoad(Xv_opaque frame, int lineno, char* line)

external access
    int MapBox::cfIsMe(char* l)
    int MapCylinder::cfIsMe(char* l)
    void Utils::cfNotice_OK(char *message)
    MapBox* MapBox::new()
    MapCylinder* MapCylinder::new()
    void MapBox::delete(MapBox*)
    void MapCylinder::delete(MapCylinder*)

```

The transition section is as follows. Note that the load(), save(), and draw() functions are not defined for MapObject. They will be specialized by each subclass.

```

Si = MapObject() --> no response
Si = next --> returns value from last set_next(n) call, otherwise returns NULL
Si = set_next(p) --> p
Si = load(f,n,l) --> not implemented in this class

```

```

Si = save(b, bs) --> not implemented in this class
Si = draw(d,xw,s) --> not implemented in this class
Si = cfSelectAndLoad(f,n,l) ∧ MapBox::cfIsMe(l) = TRUE ∧
    (p = new MapBox)->load(f,n,l) = TRUE --> p
Si = cfSelectAndLoad(f,n,l) ∧ MapBox::cfIsMe(l) = TRUE ∧
    (p = new MapBox)->load(f,n,l) = FALSE --> NULL
Si = cfSelectAndLoad(f,n,l) ∧ MapBox::cfIsMe(l) = FALSE ∧
MapCylinder::cfIsMe(l) = TRUE
    (p = new MapCylinder)->load(f,n,l) = TRUE --> p
Si = cfSelectAndLoad(f,n,l) ∧ MapBox::cfIsMe(l) = FALSE ∧
MapCylinder::cfIsMe(l) = TRUE
    (p = new MapCylinder)->load(f,n,l) = FALSE --> NULL
Si = cfSelectAndLoad(f,n,l) ∧ MapBox::cfIsMe(l) = FALSE ∧
MapCylinder::cfIsMe(l) = FALSE -->
    Utils::cfNotice_OK(f, "Map file format error: unknown object @ line <n>");

```

For MapBox, the black box header is as follows (throughout this section, MapBox will be used to show the subclass specification process; MapCylinder is nearly identical):

```

access programs
    MapObject() <inherited>
    MapObject* set_next(MapObject*) <inherited>
    virtual int load(Xv_opaque frame, int lineno, char* line) <inherited>, <overridden>
    virtual void save(char *buffer, int bufsize) <inherited>, <overridden>
    virtual void draw(Display *display, Window xid, int scale) <inherited>, <overridden>

output variables
    MapObject* next <inherited>

class access programs
    static void cfSelectAndLoad(Xv_opaque frame, int lineno, char* line) <inherited>
    static int cfIsMe(char* line)

external access
    void Utils::cfNotice_OK(char *message)

```

Note that certain functions are inherited but not overridden. In this case, these functions do not need to be redefined in the black box. The transition is as follows:

```

Si = load(f,n,l) ∧ legal_box(l) --> TRUE
Si = load(f,n,l) ∧ not(legal_box(l)) -->
    Utils::cfNotice_OK(f, "Map file format error: bad box definition @ line <n>")
    return FALSE
Si = save(b, bs) -->
    copy information from load() into "box <locx> <locy> <width> <length>
    <height>" with default height if none specified by load() and limited to length
    of bs.
Si = draw(d,xw,s) -->
    draw rectangle at <locx>*s,<locy>*s+<length>*s of size
    <width>*s,<length>*s

```

$S_i = \text{cflsMe}(l) \wedge \text{strcmp}(l, \text{"box"}, 3) = 0 \rightarrow \text{TRUE}$
 $S_i = \text{cflsMe}(l) \wedge \text{strcmp}(l, \text{"box"}, 3) \neq 0 \rightarrow \text{FALSE}$

Verify that all top-level black box stimuli are assigned to classes and that the lower-level black boxes perform all the top-level operations.

There is no top-level box for the MapObject classes, but they are constrained by the requirements set by the Map class. Therefore, it must be verified that the required functions have been included and that the syntax for these functions matches that used in the Map class.

Verify that all inter-class stimuli used in black boxes match receiving boxes' specifications.

This class hierarchy has no interactions with classes other than the creating class.

Create TAM specifications for the classes

- use black box header to create syntax section

The syntax tables for MapObject are as follows. Note that some input and output variables listed were not present in the input or output sections of the black box above. In the trace table, outputs which are returned from access program calls, or inputs that result from external function calls must be documented in the input and output tables. For example, buffer is listed in the output variables table, but maps to the buffer parameter returned from the save() access program.

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Other
MapObject	(constructor)				
set_next	<MapObject*>	<MapObject*> nextobj			
load	<int> load_ok	<Xv_opaque> frame	<int> lineno	<char*> line	virtual
save	<void>	<char*> buffer	<int> bufsize		virtual
draw	<void>	<Display*> display	<Window> xid	<int> scale	virtual

OUTPUT VARIABLES

Variable Name	Type	Access
next	<MapObject*>	public
load_ok	<int>	fn return
buffer	<char*>	fn param return

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3
cfSelectAndLoad	<MapObject*> created	<Xv_opaque> frame	<int> lineno	<char*> line

CLASS OUTPUT VARIABLES

Variable Name	Type	Access
created	<MapObject*>	fn return

INPUT VARIABLES

Variable Name	Type	Access
boxnew	<MapBox*>	ext fn return
cylnew	<MapCylinder*>	ext fn return

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
MapBox::cfIsMe	<int>	<char*> line	
MapCylinder::cfIsMe	<int>	<char*> line	
Utils::cfNotice_OK	<void>	<Xv_opaque> frame	<char*> message
MapBox::new	<MapBox*> boxnew		
MapCylinder::new	<MapCylinder*> cylnew		
MapBox::delete	<void>	<MapBox*>	
MapCylinder::delete	<void>	<MapCylinder*>	

For MapBox, the syntax tables are as follows (note that items marked with (i) are inherited):

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Other
(i) MapObject	(constructor)				
(i) set_next	<MapObject*>	<MapObject*> nextobj			
(i) load	<int> load_ok	<Xv_opaque> frame	<int> lineno	<char*> line	virtual, overridden
(i) save	<void>	<char*> buffer	<int> bufsize		virtual, overridden
(i) draw	<void>	<Display*> display	<Window> xid	<int> scale	virtual, overridden

OUTPUT VARIABLES

Variable Name	Type	Access
(i) next	<MapObject*>	public
(i) load_ok	<int>	fn return (overridden)
(i) buffer	<char*>	fn param return
(output _screen)	(X display window)	N/A

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3
(i) cfSelectAndLoad	<MapObject*> created	<Xv_opaque> frame	<int> lineno	<char*> line
cfIsMe	<int> isMe	<char *> line		

CLASS OUTPUT VARIABLES

Variable Name	Type	Access
(i) created	<MapObject*>	fn return

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
Utils::cfNotice_OK	<void>	<Xv_opaque> frame	<char*> message

- create canonical trace using access programs and input events

For the MapObject class, the only real output is the next variable. Therefore, the only information that needs to be included in the canonical trace is the information required to give the value for this variable. For the class trace, the output needed is the pointer to a new object created from the cfSelectAndLoad() function. This value will actually come from an external function call, so this information had to be included in the input variables table for the class so it could be included in the class canonical trace. The canonical traces for MapObject are:

$$\text{canonical}(T_i) \leftrightarrow (T_i = \text{MapObject}()) \vee (T_i = \text{set_next}(n))$$

$$\text{canonical}(T_c) \leftrightarrow (T_c = _) \vee (T_c = \text{boxnew}) \vee (T_c = \text{cylnew})$$

The MapBox object adds the load(), save() and draw() functions. The save() function does not affect the output of the class and does not have to be included in the canonical trace. The load() function must be included since its parameters determine the details of the item's form. The draw() function must be included to ensure that the screen output is handled correctly. On the class side, there is a new output for the cfIsMe function, based

on its parameters, so it must be included in the class canonical trace. The canonical traces for this class are:

$$\begin{aligned} \text{canonical}(T_i) \leftrightarrow & (T_i = \text{MapObject}() \vee \text{set_next}(n)) \vee \\ & (T_i = [\text{MapObject}() \vee \text{set_next}(n)].\text{load}(f, \text{ln}, l)) \vee \\ & (T_i = [\text{MapObject}() \vee \text{set_next}(n)].\text{load}(f, \text{ln}, l).\text{draw}(d, \text{xw}, s)) \end{aligned}$$

$$\text{canonical}(T_c) \leftrightarrow (T_c = _) \vee (T_c = \text{boxnew}) \vee (T_c = \text{cylnew}) \vee (T_c = \text{cfIsMe}(l))$$

- create equivalences based on canonical trace and output requirements; create auxiliary functions and dictionary entries as needed

The equivalences for MapObject are relatively simple, with the exception of the equivalence for cfSelectAndLoad(), which is complex because of its interaction with functions of the subclasses. The equivalences are:

$T.\text{MapObject}() \equiv \text{MapObject}()$

$T.\text{set_next}(n) \equiv \text{set_next}(n)$

$T.\text{load}(f, \text{ln}, l) \equiv \% \text{undefined for this class} \%$

$T.\text{save}(b, bs) \equiv \% \text{undefined for this class} \%$

$T.\text{draw}(d, \text{xw}, s) \equiv \% \text{undefined for this class} \%$

$T_C.cfSelectAndLoad(f, ln, l) \equiv$

conditions	equivalences
$MapBox::IsMe(l)=TRUE \wedge$ $(boxnew = new MapBox) \rightarrow load(f, ln, l)=TRUE$	$equiv = boxnew;$ $ADD-TO-TRACE(T_{cmb}, IsMe(l));$ $ADD-TO-TRACE(T_{cmb}, new);$ $ADD-TO-TRACE(T_{boxnew}, load(f, ln, l));$ where T_{cmb} is the class trace for MapBox
$MapBox::IsMe(l)=TRUE \wedge$ $(boxnew = new MapBox) \rightarrow load(f, ln, l)=FALSE$	$equiv = _;$ $ADD-TO-TRACE(T_{cmb}, IsMe(l));$ $ADD-TO-TRACE(T_{cmb}, new);$ $ADD-TO-TRACE(T_{boxnew}, load(f, ln, l));$ $ADD-TO-TRACE(T_{cmb}, delete);$ where T_{cmb} is the class trace for MapBox
$MapBox::IsMe(l)=FALSE \wedge$ $MapCyl::IsMe(l)=TRUE \wedge$ $(cylnew = new MapBox) \rightarrow load(f, ln, l)=TRUE$	$equiv = cylnew;$ $ADD-TO-TRACE(T_{cmb}, IsMe(l));$ $ADD-TO-TRACE(T_{cmc}, IsMe(l));$ $ADD-TO-TRACE(T_{cmc}, new);$ $ADD-TO-TRACE(T_{cylnew}, load(f, ln, l));$ where T_{cmb} is the class trace for MapBox and T_{cmc} is the class trace for MapCylinder
$MapBox::IsMe(l)=FALSE \wedge$ $MapCyl::IsMe(l)=TRUE \wedge$ $(cylnew = new MapBox) \rightarrow load(f, ln, l)=FALSE$	$equiv = _;$ $ADD-TO-TRACE(T_{cmb}, IsMe(l));$ $ADD-TO-TRACE(T_{cmc}, IsMe(l));$ $ADD-TO-TRACE(T_{cmc}, new);$ $ADD-TO-TRACE(T_{cylnew}, load(f, ln, l));$ $ADD-TO-TRACE(T_{cmc}, delete(cylnew));$ where T_{cmb} is the class trace for MapBox and T_{cmc} is the class trace for MapCylinder
else	$equiv = _;$ $ADD-TO-TRACE(T_{cmb}, IsMe(l));$ $ADD-TO-TRACE(T_{cmc}, IsMe(l));$ $ADD-TO-TRACE(T_u, cfNotice_OK(f, "Map file format error: unknown object @ line <n>"));$ where T_u is the class trace for Utils, T_{cmb} is the class trace for MapBox and T_{cmc} is the class trace for MapCylinder

$T_C.boxnew \equiv boxnew$

$T_C.cylnew \equiv cylnew$

For MapBox, the equivalences are also relatively simple. Note that the equivalence for `set_next()` is included although it is not being overridden. This is because the equivalence in MapObject does not make sense for the subclasses, and it seemed to make sense to include a new equivalence for it in this subclass to make clear its effect on the canonical trace.

An auxiliary function is defined to assist in parsing the MapBox trace:

parse(S,S1,S2,S3) =

conditions	equivalences
(S = S1.S2.S3.S4) $\wedge$ (S1 = [MapObject() $\vee$ set_next(n)]) $\wedge$ (S2 = [load(f,ln,l)] _{i=0} ¹) $\wedge$ (S3 = [draw(d,xw,s)] _{i=0} ¹)	true
else	false

The equivalences for MapBox are as follows:

T.set_next(n) $\equiv$ set_next(n).L.D where parse(T, I, L, D)

T.load(f, ln, l) $\equiv$

conditions	equivalences
legal_box(l)	I.load(f, ln, l) where parse(T, I, L, D)
else	equiv = I.load(f, ln, l) where parse(T, I, L, D); ADD-TO-TRACE(T _U , cfNotice_OK(f, "Map file format error: bad box definition @ line <ln>") where T _U is the class trace for Utils

T.save(b, bs) $\equiv$ T

T.draw(d, xw, s) $\equiv$

conditions	equivalences
parse(T, I, L, D) $\wedge$ L = load(f, ln, l) $\wedge$ legal_box(l)	I.L.draw(d, xw, s)
else	%cannot draw without legal load() first%

T_C.cfIsMe(l) $\equiv$ cfIsMe(l)

- create outputs based on canonical trace

As mentioned above, for MapObject, the only current instance output is the next variable, which is NULL or the value from the set_next() call in the canonical trace. Other instance outputs, while mentioned in MapObject, are undefined in the superclass. The class output from cfSelectAndLoad is set according to the canonical trace as shown.

OUTPUT VALUES

V[next](T) =

conditions	values
parse(T, I, L, D) $\wedge$ I = set_next(n)	n
else	NULL

V[load_ok](T) = %undefined%

V[buffer](T) = %undefined%

V[created](T_c) =

conditions	values
T _c = boxnew	value of boxnew
T _c = cylnew	value of cylnew
else	NULL

RETURN VALUES

Program Name	Argument No	Value
cfSelectAndLoad	Value	created

For MapBox, as noted, V[next] and V[created] are unchanged from the superclass. Note that while load_ok is a simple boolean result from a function to indicate its success or failure, information must be contained in the canonical trace for this output to be valid. This is true even if the information required in the canonical trace would have no meaning beyond the scope of this function.

OUTPUT VALUES

V[load_ok](T) =

conditions	values
parse(T, I, L, D) ∧ L=load(f,ln,l) ∧ legal_box(l)	TRUE (1)
parse(T, I, L, D) ∧ L=load(f,ln,l) ∧ not(legal_box(l))	FALSE (0)
else	%undefined%

V[buffer](T) =

conditions	values
parse(T, I, L, D) ∧ L=load(f,ln,l) ∧ legal_box(l)	"box <locx> <locy> <width> <length> <height>" from load() with default height if none specified
else	""

V[(output_screen)](T) =

conditions	values
parse(T, I, L, D) ∧ L=load(f,ln,l) ∧ legal_box(l) ∧ D=draw(d,xw,s)	draw rectangle parsed from l in window defined by d, xw with scale s
else	%undefined%

$V[isMe](T_C) =$

conditions	values
$T_C = cfIsMe(l) \wedge \text{strcmp}(\text{"box"}, l, 3) = 0$	TRUE (1)
$T_C = cfIsMe(l) \wedge \text{strcmp}(\text{"box"}, l, 3) \neq 0$	FALSE (0)
else	%undefined%

RETURN VALUES

Program Name	Argument No	Value
(i) load	Value	load_ok (overridden)
(i) save	Arg#1	buffer (overridden)
(i) cfSelectAndLoad	Value	created
cfIsMe	Value	isMe

- verify specification using Parnas Verification Checklist.

Verify TAM specification to black boxes for each class.

The verification is as mentioned for the example given in the previous section. The verification information accompanies the specification contained in Appendix D.

Write C++ header for each object using TAM specification and verify to specification.

The C++ header is created first from the TAM specification. Private and protected items for storage by the object may be added as the code is being developed. Protected means that the items are visible to subclasses but not to external classes.

The C++ header for MapObject is as follows:

```
class MapObject {
protected:
    double locx, locy, height;

public:
    MapObject();
    MapObject* set_next(MapObject* nextobj);
    virtual int load(Xv_opaque frame, int lineno, char* line) = 0;
    virtual void save(char* buffer, int bufsize) = 0;
    virtual void draw(Display* display, Window xid, int scale) = 0;

// class functions
    static MapObject* cfSelectAndLoad(Xv_opaque frame, int lineno,
                                      char* line);
};
```

The load(), save(), and draw() are not implemented and are therefore written as 'pure virtual functions', in C++ terminology, meaning that they are not implemented in the abstract base class.

For MapBox, the C++ header is as follows:

```
class MapBox : public MapObject {  
// private  
    double width, length;  
  
public:  
// class functions  
    static int cfIsMe(char* line);  
};
```

The 'public MapObject' line means that it is inheriting from MapObject and keeping the public members of MapObject public in this subclass. MapBox is largely empty since the public interface for the subclass was mostly defined by the superclass. The private entries differ due to the information that must be stored to represent a box (length and width). Note that the information common to both boxes and cylinders, namely location and height, are contained in the MapObject definition.

Create C++ objects and main()

- write using C++ header and TAM specification

The operation of the C++ functions is as mentioned in the original requirements for MapObject.

- verify versus TAM specification and C++ header

- verify cross-object access is correct.

Verification is as mentioned in the previous example.

Any lower-level C++ classes "discovered" should be developed by creating a black box definition for the class and then designing as described above.

One additional function was identified in developing the C++ code, called `nextfield()`, which assists in parsing the lines read in from a file. Its specification is not included here.

5 Conclusion

This project was undertaken to better understand the Trace Assertion Method and to see if it was a useful specification method for working with object-oriented design. Notational changes have been added to the method to handle specific elements of object-oriented systems, including class versus instance, constructors/destructors, and inheritance. In addition, the method, which is designed for module specification, has been integrated into a larger process for complete system specification, development, and verification.

The results are mixed. TAM provides some useful representations for dealing with modular code, and adapting it for object-oriented design was not extremely difficult, but our interpretation of it does produce a voluminous amount of specification material, and can lead to convoluted specifications in order to meet the TAM requirements.

TAM provides a body of important information for the designer in a well organized format, including a clear idea of the input and output variables and their types, access programs and their syntax, events of interest—i.e. state—via the canonical trace, and values and window of validity for outputs. The organization of this information into tables and the use of a specific structure makes it easy for the designer to find the appropriate information in a TAM specification. The information maps directly into a C++ header definition for a class. In addition, there are specific steps for verification of the tables which allow a designer to immediately check whether a table is complete and consistent with other pieces of the specification.

On the other hand, while TAM gives a good idea of how the various functions of the module/class interact—such as the equivalences and the canonical trace—it does not give a clear view of what each access program actually does. A black box description seems superior in that regard. The concern is whether the TAM description communicates information to the implementor in a useful form. It seems more oriented toward assisting the designer/specifier to ensure the specification is complete, and less toward helping the programmer understand how the code must perform. Nevertheless, it

is the responsibility of the programmer to learn to read these specifications in order to make this process easier.

Another problem is that simple stimulus-response pairs for an object can result in a tangled canonical trace in order to hold certain program state information. For example, to display XView Notice outputs, a new event must be in the canonical trace to indicate exactly when such a notice appears. In addition, to represent simple return values from functions requires that information appear in the canonical trace which is otherwise unnecessary. In effect, putting this information in the canonical trace makes it more difficult to decipher and develop, and makes it more complicated than the clear box that will follow.

Finally, the method developed here produces large amounts of specification material for a simple system. A good example of this is the equivalence for the save() function in Map. The equivalence table for this is long, since it is completely non-procedural. The code itself is much simpler since handling the cases is simpler when considered procedurally. While some might argue that more is better, this creates problems in terms of keeping a large body of material under proper revision control and insuring that verification between various steps is completed properly.

The modified method and process presented here may be applicable for critical systems, for those requiring more care and documentation, or for those involving a large number of interacting objects. In these cases, the additional steps required under this method will be worth the effort in order to better understand and document the system being developed.

Bibliography

- [Bartussek78] Bartussek, W. and D.L. Parnas, "Using Assertions about Traces to Write Abstract Specifications for Software Modules," *Proceedings of Second Conference of European Cooperation in Informatics, Lecture Notes in Computer Science*, 1978. 65, Springer Verlag, p. 211-236.
- [Booch91] Booch, G., *Object Oriented Design: With Applications*. 1991, Benjamin/Cummings.
- [Casais93] Casais, E., *et al.*, "Formal Methods and Object-Orientation," Tutorial at TOOLS Europe, Versailles, France, 1993.
- [Cook92] Cook, W.R., "Interfaces and Specifications for the Smalltalk-80 Collection Classes," *Proceedings of OOPSLA '92*, Vancouver, BC, 1992.
- [Duke91] Duke, R., *et al.*, "The Object-Z Specification Language: Version 1," Technical Report No. 91-1, Software Verification Research Centre, Dept. of Computer Science, The Univ. of Queensland, May 1991.
- [Erskine92] Erskine, N.S., "The Usefulness of the Trace Assertion Method for Specifying Device Module Interfaces," CRL Report No. 258, Telecommunications Research Institute of Ontario, McMaster University, August 1992.
- [Erskine93] Erskine, N., Personal Communication, April 19, 1993.
- [Gehani86] Gehani, N. and A. McGettrick, ed. *Software Specification Techniques*. 1986, Addison-Wesley.
- [Henricson92] Henricson, M. and E. Nyquist, "Programming in C++: Rules and Recommendations," Document No. M 90 0118 Uen, Ellemtel Telecommunication Systems Laboratories, Oct. 10, 1992.
- [Hevner93] Hevner, A.R. and H. Mills D., "Box Structured Methods for Systems Development with Objects," *IBM Systems Journal*, 1993. (To Appear).
- [Hoffman88] Hoffman, D. and R. Snodgrass, "Trace Specifications: Methodology and Models," *IEEE Transactions on Software Engineering*, 1988. 14(9): p. 1243-1252.
- [Hoffman89] Hoffman, D., "Practical Interface Specification," *Software—Practice and Experience*, 1989. 19(2): p. 127-148.
- [Koenig89] Koenig, A., *C Traps and Pitfalls*. 1989, Addison-Wesley.
- [Leavens92] Leavens, G.T. and Y. Cheon, "Preliminary Design of Larch/C++," U. Martin and J.M. Wing ed., *First International Workshop on Larch*, July 1992.
- [McLean84] McLean, J., "A Formal Method for the Abstract Specification of Software," *Journal of the Association for Computing Machinery*, 1984. 31(3): p. 600-627.
- [Meira90] Meira, S.R.L. and A.L.C. Cavalcanti, "Modular Object Oriented Z Specifications," J.E. Nicholls ed., *Z User Workshop: Proceedings of the Fourth Annual Z User Meeting*, Oxford, 1990.
- [Meira92] Meira, S.R.L. and A.L.C. Cavalcanti, "The MooZ Specification Language Version 0.4," *Relatorio Tecnico ES/1.92*, Universidade Federal de Pernambuco, Departamento de Informatica, Recife-PE, January 1992.
- [Meyer90] Meyer, B., *Object-Oriented Software Construction*. 1990, Prentice Hall.
- [Mills86] Mills, H.D., R.C. Liger, and A.R. Hevner, *Principles of Information System Analysis and Design*. 1986, Academic Press, Inc.

- [Mills87a] Mills, H.D., M. Dyer, and R.C. Linger, "Cleanroom Software Engineering", *IEEE Software*. September 1987, p. 19-24.
- [Mills87b] Mills, H.D., R.C. Linger, and A.R. Hevner, "Box structured information systems," *IBM Systems Journal*, 1987. 26(4): p. 395-413.
- [Mills88] Mills, H.D., "Stepwise Refinement and Verification in box-structured systems", *IEEE Computer*. June 1988, p. 23-26.
- [Parnas72] Parnas, D.L., "A Technique for Software Specification with Examples," *Comm. of the ACM*, 1972. 15(5): p. 330-336.
- [Parnas89] Parnas, D.L. and Y. Wang, "The Trace Assertion Method of Module Interface Specification," Technical Report 89-261, TRIO, Queens University, Kingston, Ontario, 1989.
- [Rose92] Rose, G., "Object-Z", in *Object Orientation in Z*, S. Stepney, R. Barden, and D. Cooper, Editor. 1992, Springer-Verlag. p. 59-77.
- [Stepney92] Stepney, S., R. Barden, and D. Cooper, ed. *Object Orientation in Z*. Workshops in Computing, 1992, Springer-Verlag.
- [Trammell93] Trammell, C., *SQRL Verification Questions*, 1993.
- [Wegner90] Wegner, P., "Concepts and Paradigms of Object-Oriented Programming", *Object-Oriented Messenger*. September 1990.
- [Wirfs-Brock90] Wirfs-Brock, R., B. Wilerson, and L. Wiener, *Designing Object-Oriented Software*. 1990, Prentice Hall.

Appendices

Appendix A: An Example Specification

SAVINGS TRACKER MODULE

TYPE IMPLEMENTED: <Savings>

(1) SYNTAX

OUTPUT VARIABLES

Variable Name	Type
balance	<float>

ACCESS PROGRAMS

Program Name	Value	Arg#1
DEPOSIT		<float>
WITHDRAW		<float>
BALANCE	<float>	

(2) CANONICAL TRACES

$$\text{canonical}(T) \leftrightarrow (T = [\text{DEPOSIT}(x_i)]_{i=0}^n . [\text{WITHDRAW}(y_j)]_{j=0}^m)$$

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

AUXILIARY FUNCTIONS

Func Name	Value	Arg#1	Arg#2	Arg#3
parse	<boolean>	<trace>	<trace>	<trace>

parse(S,S1,S2) =

conditions	equivalences
$(S = S1.S2) \wedge$ $(S1 = [\text{DEPOSIT}(x_i)]_{i=0}^n) \wedge$ $(S2 = [\text{WITHDRAW}(y_j)]_{j=0}^m)$	true
else	false

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

(3) EQUIVALENCES

Completeness (1): There is one equivalence function for each event class.

- There is one each for DEPOSIT, WITHDRAW, and BALANCE.

$T.DEPOSIT(x) \equiv D.DEPOSIT(x).W$ where $parse(T, D, W)$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- no partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T defined by LHS, D & W defined by T from $parse()$.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- D is string of deposits; D.DEPOSIT.W maintains canonical trace structure.

Consistency (3): All RHC values are unique:

- One value, therefore unique.

$T.WITHDRAW(y) \equiv$

conditions	equivalences
$\sum_{i=0}^n x_i + \sum_{j=0}^m y_j \geq y$ where $parse(T, D, W) \wedge D = [DEPOSIT(x_i)]_{i=0}^n \wedge$ $W = [WITHDRAW(y_j)]_{j=0}^m$	$T.WITHDRAW(y)$
else	%insufficient funds%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T defined by LHS

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T is canonical by definition.

Consistency (3): All RHC values are unique:

- One value, one error.

$T.BALANCE \equiv T$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- no partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T defined by LHS.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T is canonical by definition.

Consistency (3): All RHC values are unique:

- One value.

(4) VALUES

OUTPUT VALUES

$$V[\text{balance}](T) = \sum_{i=0}^n x_i + \sum_{j=0}^m y_j$$

$$\text{where } \text{parse}(T, D, W) \wedge D = [\text{DEPOSIT}(x_i)]_{i=0}^n \wedge W = [\text{WITHDRAW}(y_j)]_{j=0}^m$$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T defined by LHS, rest are defined from T according to canonical trace.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

RETURN VALUES

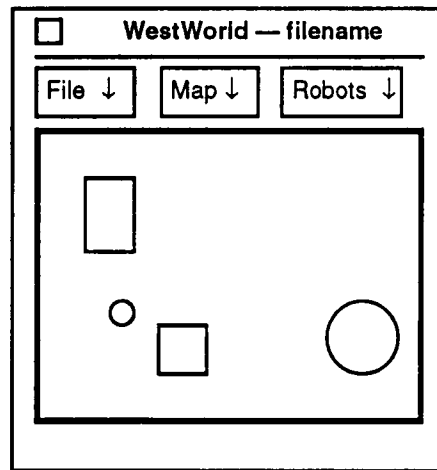
Program Name	Argument No	Value
BALANCE	Value	balance

Completeness (2): There is one output function/relation that specifies each output value:

- There is one output value $V[\text{balance}]$ defined above for the one value in the table.

Appendix B: The Requirements Document

WestWorld



Interface Requirements Document

Alex L. Bangs
Oak Ridge National Laboratory
Center for Engineering Systems Advanced Research

Table of Contents

I. Introduction and Overview	61
A. A Simulation of Cooperating Mobile Robots	61
B. An Examination of Other Simulation Systems	62
C. Mapper: A Prototype for the Interface	63
D. Overview: Interface Requirements	64
II. The Interface Requirements Specification	66
A. User Interaction and GUI Specification	66
1. Program Invocation	66
2. Initial Program Actions	66
3. Program Menus and Actions	66
a. The File Menu	66
b. The Map Menu	69
c. The Robots Menu	71
4. Mouse Actions	71
5. X Events/Window Manager Actions	73
B. HELIX Interface and Interactions	73
1. HELIX Events	73
2. HELIX Shared Memory	74
a. Robot/Simulator Interface	74
b. Add-On Interface	75
C. Supporting Specifications	75
1. Map Format	75
2. System Interfaces	76
a. File System (FS)	76
b. Memory Management (MM)	76
D. Summary of GUI ELEMENTS	77
III. Incremental Development Plan	78
Subappendix A: The XView Toolkit	79
Subappendix B: Glossary	81
Subappendix C: References	82

I. Introduction and Overview

A. A SIMULATION OF COOPERATING MOBILE ROBOTS

The reasons for constructing a simulator for conducting experiments in cooperation are multi-fold. First, a simulator will allow more agents to be simulated than would be practical in a laboratory environment; at least 10-20 robots is desired. Currently in CESAR we have three robots, and may purchase more in the near future, but having as many as 10 or more robots is not realistic. Second, it allows testing of some elements of cooperation, such as communications protocols and task planning algorithms without dealing with "real" robot problems. These problems include operating multiple sets of sonar sensors in the same room, setup time for getting all software loaded on robots, locating robots at desired starting positions, having operators on hand to monitor each robot, etc. While experimentation with real robots is essential, it can be done in a later phase once the basic infrastructure and algorithms for cooperation have been developed.

The basic architecture of the simulator is to be based on HELIX. HELIX is a system developed in CESAR to provide a communications system for processes running on a heterogeneous network of systems [Jones92a; Jones92b]. This system allows both shared-memory and message-passing communications. Since our current base of robot code runs under HELIX, it makes sense to base the simulator and the experimental element of the work on HELIX since it will allow use of current code as well as make the integration of our system easier.

The main process of the simulator will be a graphical user interface that will show the progress of the various robots in the environment via animation, and will display related data (sensor sweeps, confidence maps, communicated data, etc.). The simulation will be linked into the simulations of the various robots via HELIX. Using HELIX will allow the use of both simulated and real robots during a simulation run. This ability to display real robot status will make the interface useful as a console for a multirobot operation.

The simulation of each individual robot simulator could vary according to the complexity of the simulation. The general design of a robot simulator using HELIX would use a separate process to handle the cooperative aspects of control, while other processes would handle the other functions of robot control. In a simple simulation, these processes could all be rolled into one process. For a more complex robot, a large number of processes might be required.

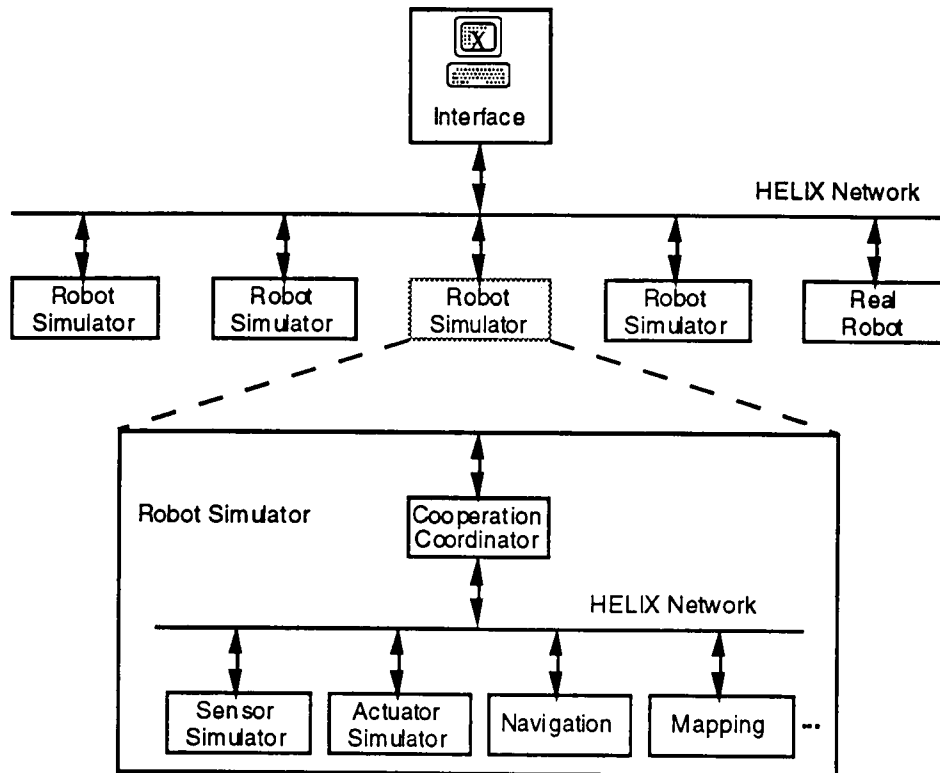


Fig.1: Interface and Simulation Components

Since each of these separate robots is represented by a cooperation process, the view of the whole system as seen by each robot or the interface is basically a collection of cooperation processes. Figure 1 shows one possible simulation setup. Note that also a component could be a real robot rather than a simulation. The HELIX network inside each component is separate from the network between components. This will be handled by a special version of HELIX called N-HELIX, to be developed.

Construction of the complete simulation system will involve not only building the user interface, but also the simulated robots (including simulations for actuators and sensors), simulated "worlds", and the processes required for cooperation. This document focusses specifically on the requirements for the interface portion of the simulation. The contents include a discussion of prototypes of the interface already developed, the role of the interface in the system, a breakdown of requirements between the interface and add-on components, and the requirements specification of the interface, and an incremental development plan.

B. AN EXAMINATION OF OTHER SIMULATION SYSTEMS

Several other simulation systems have been examined, both to determine if they were up to the task of cooperating robot simulations as well as to note features for possible inclusion into the simulator being developed.

First, a few simulations capable of working with multiple robots have been constructed. A simulator is being developed at RIKEN specifically to support work there in multirobot cooperation [Habib92]. It currently consists of a simple system capable of displaying an environment map and the robot movements. The system runs on a Silicon Graphics machine and uses IPC to communicate between the display process and the robot simulation processes. As of 7/92, a separate ultrasonic simulation had been completed, but was still to be integrated into the full simulation [Asama92].

Another simulation capable of supporting multiple robots is one being used by Lynne Parker at MIT. This simulator, originally developed by Yasuo Kagawa of Mazda while visiting MIT, runs in Common Lisp on a Macintosh, and has been extended by Ms. Parker. Due to running on a Macintosh and under lisp, its operation is slow [Parker92b; Parker92c].

Commercial simulation packages, such as IGRIP, are undesirable because of their cost and limitations (such as the local limitation that it can only be run on one machine).

Several simulations that have been examined might be capable of multirobot work with some adaptation. EDDIE, from CMU, is purported to be capable of multirobot simulation, but a workstation is required for each robot to be simulated, thus limiting its ability to scale to large populations of robots [Gowdy91; Parker92b].

Yutaka Watanabe has developed a sophisticated simulation system at ORNL which is specially designed to work with an omnidirectional platform also developed at ORNL. He is planning to extend this simulator to be capable of handling more than one robot at a time. The simulation is feature-packed, with a large number of display options available. These include the usual ability to monitor the position and orientation of the robot as well as the wheel orientations on the platform; a map display with a grid; fuzzy rule evaluation; vision, which shows a 3D perspective view of the robot based on the world model; sonar, which shows the sonar beacons scanning on the map; and an option to display the simulation's "real" position for the robot vs. the estimated position kept by the user program running on the robot. He also has a tool that allows the map to be dumped out and edited by idraw for printing or inclusion into other documents [Watanabe93].

Another simulator is the one that has been developed for use in the robotics lab at Tsukuba University with their small robots called Yamabico. This simulator is for single robots only [Kimoto92a]. A large amount of time was invested in developing the sonar simulation for this system, including two basic types of reflection which can be chosen—a simple model and a more complex model using diffusion. A good feature of this simulator is that it was developed to run programs written for the robot without modification [Kimoto92b].

Finally, a simulation has been developed at the University of Michigan for experimenting with distributed AI problems. This simulation is Common Lisp based [Mont90]. Having this Lisp-based makes it more difficult to integrate with CESAR systems.

[Torrance92] provides a useful discussion of simulation problems and benefits.

C. MAPPER: A PROTOTYPE FOR THE INTERFACE

A very simple prototype of an interface was initially written in Smalltalk. While this prototype was not very sophisticated, it did give some ideas on object-oriented programming and interface elements.

A second prototype called "mapper" was written, using C++ and HELIX. It is a very simple program for loading object maps and displaying robots moving in the map environment. The robots are controlled by separate processes which give their position to the mapper program via HELIX shared memory, and send and receive events from the mapper.

The GUI was built using a Sun tool called "GUIDE" which allows interactive building of an interface, and dumps out XView graphics code and call-back stubs (C++ in this case) which can then be modified by the programmer.

Some observations about the mapper interface:

- mapper allows only one map to be loaded at a time. If a new map is loaded, it destroys all the robots currently in the map, but does not ask the user if this is ok, nor does it send quit messages to the affected robot simulators.
- mapper only allows one coordinate system, i.e. the current map, to be worked on; it does not handle multiple spaces or multiple windows. This is probably sufficient for most simulations. If a large complex space needs to be simulated, it can be done all in one map.
- mapper creates an event menu for each new robot to allow individual control of that robot. This does not scale well to many robots, however, since there is not enough menu-bar space. Therefore individual robot actions will have to be handled both or either through a click-on-the-robot type interface to pop up a menu, or through a command-line/dialog box interface where the robot number is specified. While the current version has a menu to send messages to all robots, this is also probably not good for many robots. Thus a grouping scheme for robots would be useful, and then use a dialog interface to send an event to all members of a specific group.
- when a robot is created, the user locates and orients the robot with the mouse. This will have to be made optional, to be selected by the simulator. For large numbers of robots in a simulation, manual placement may be unwieldy.
- mapper adjusts the map to fit the window size; this should be improved to first adjust the map to the window, then adjust the window to fit exactly to the map. This will look better.
- when a map is large enough that it scaling it will go beyond the minimum size for map magnification, then the mapper window should be scalable to accommodate the larger size.

Some internal/coding observations:

- XView objects as built by GUIDE are awkward and basically structs
- The mapper was not very object-oriented; a new object was added later, RobotX, which made the design more object-oriented. Some effort was made to design this object with box structures.

D. OVERVIEW: INTERFACE REQUIREMENTS

The interface has two primary roles: display and control. For display, the user should be able to see the locations of the robots and the objects in the simulated world. The user should also be able to focus on specific items of a robot's status, including position, velocity, sensor status, and internal representation information. For control, the user should be able to (a) create new simulated robots and place them in the environment, (b) delete robots from the environment, (c) start and stop robots in the environment, (d) have some teleoperation capabilities, i.e. directing a robot toward a new goal, etc.

In addition to manipulating robots, the user should be able to manipulate the virtual world in which the robots exist. The user should be capable of creating, changing, moving, and deleting objects in the map as well as changing the dimensions of the map. In addition, loading and saving map files (which are text files using a special object description language), and allowing maps to be loaded on top of one another, as well as allowing certain maps to be known a-priori by the robots, while other maps are not known are all desired.

The interface will have two primary modes of interaction: with the user via the GUI, and with the cooperating components. The GUI will be done in XView (an X windows toolkit), and the graphics code should be as isolated as possible to permit porting to another window system (see Subappendix A for an explanation of why XView was chosen). The system will use HELIX as its communications system for cooperation, in order to allow connectivity to existing robot code.

In addition, the interface will probably need to be split into multiple processes to allow new simulation-specific components ("add-ons") to be added later without changing the interface internals. For example, if we want to add a capability to the simulator to display a sensor map in a window, it makes the most sense to put this into a separate process that maintains its own window and communicates directly with the sensing process. Therefore, the interface will have to have yet another mode of communication, to add-on elements. Communication between interface elements should be done via HELIX.

Given this ability to extend the interface with add-ons, the primary interface element should include the basics from above. This includes the basic control functions, i.e. create/delete/start/stop/relocate/restart, and basic display functions, i.e. display map, robots, basic status info (position, velocity). Abilities beyond this should be developed in add-ons. This includes displays such as internal and sensor information and special control functions. Where possible, the interface should be designed to easily accomodate new robots or simulators with little or no modifications to the core interface.

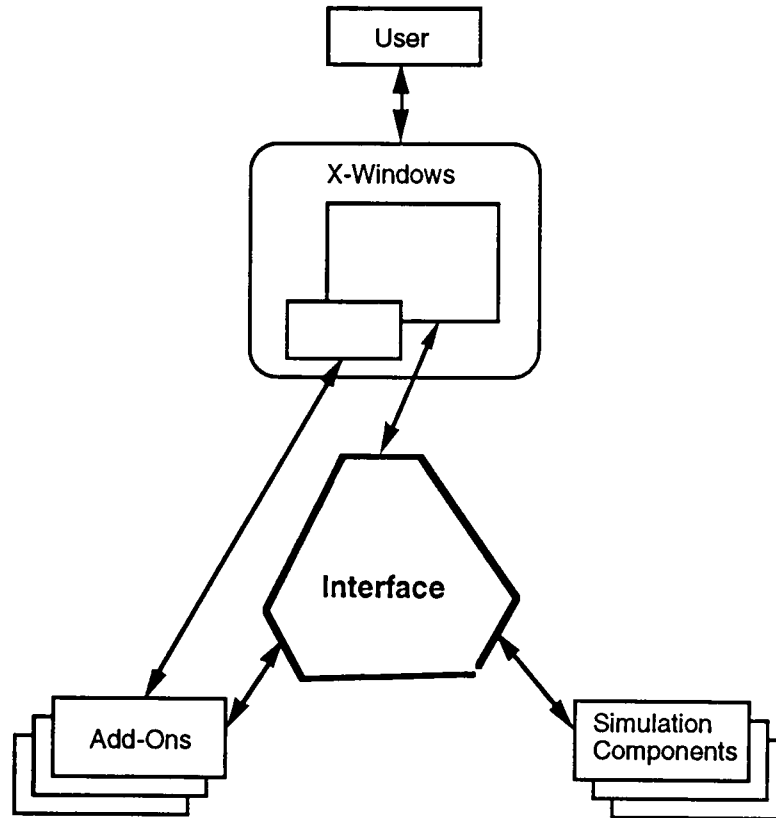


Fig. 2: Connections to the Interface

Not all the functions desired of the interface have to be provided in the first increment. In fact, the add-ons are perfect for later increments, as are some of the advanced map handling and event passing features described above. An increment plan is presented at the end of this document.

II. The Interface Requirements Specification

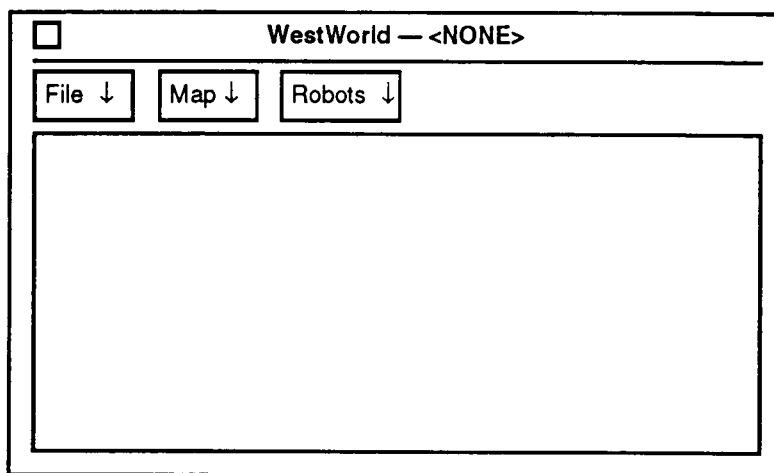
A. USER INTERACTION AND GUI SPECIFICATION

1. Program Invocation

On invocation, the user can optionally give a filename of an initial map to the program. Any errors on loading this initial map are reported as below. Other than this, the program does not have any specific command line options. It should, however, accept and pass to XView any X or XView options from the command line. See the xview man page [Sun91] for a list of XView options.

2. Initial Program Actions

The initial screen presented by the program looks like this (see Subappendix A for an explanation of XView graphical elements):



[WinMap_Empty]

This screen should be sized so that the default map size, 12.8 x 12.8m, fits the canvas exactly using a pre-selected scale (see below for information on map size and scale).

The <NONE> indicates that no map file has been loaded. If there is a problem setting up X, then an appropriate error is written on the invoking terminal and the program terminates.

3. Program Menus and Actions

a. The File Menu

The File menu consists of the following options:

- Load...
- Save...
- Quit [default]

Selection of the *Load...* option presents the following pop-up box:

[PopupLoadMap]

Filename is set to the last file loaded or the last file saved (whichever is most recent) or blank.

The sideways (unpinned) pin in the header means that the box will disappear when the Load button is pressed, unless the pin is pressed, in which case it will stay on the screen. If the pin is pressed again, the box will disappear, effectively canceling the Load... command.

If the file cannot be opened, then the following notice appears:

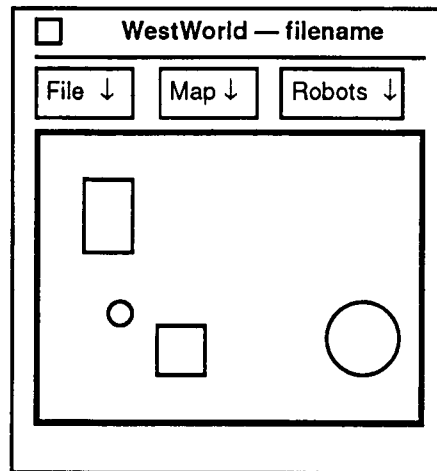
[NoticeOK_FileOpenErr]

Other filesystem errors that will be reported are "An error occurred reading the file", "An error occurred writing the file", and "An error occurred closing the file". If the file can be opened but there is an error the data format of the file, this error is reported as verbosely as possible, to allow the user to correct the error in the file.

[NoticeOK_FileFormatErr]

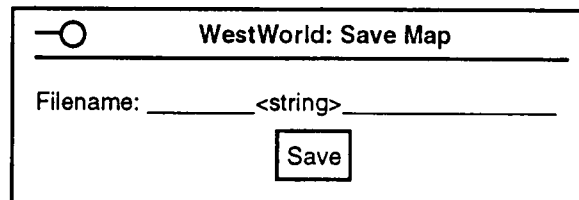
See "Map Format" for details on the correct format for the map files. For either of these errors, the Load box remains on the screen again to allow the user another try.

If the file is correctly loaded, the new map objects plus any previously loaded are drawn on the screen to scale. Note that the map origin is its lower left-hand corner. The objects are scaled (to the integer part of a fractional scale) based on a default scale which can be changed by the user (see below). Invisible objects are drawn with dashed borders. The header is updated to show the filename (last component of path only) of the latest map file loaded. Finally, once the scale has been determined by the current screen size, the screen is re-sized to fit the map outline exactly. This gives us a window as below.



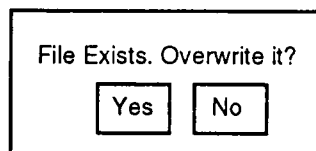
[WinMap_WithObj]

Selection of *Save...* brings up a box as with the *Load...* command. Again, this box can be disposed of by pressing on the pin. The filename is set to the last file saved or the last file accessed via load (whichever is most recent) or blank.



[PopupSaveMap]

When the *Save* button is pressed, the file is checked to see if it already exists. If so, the notice box shown is displayed.

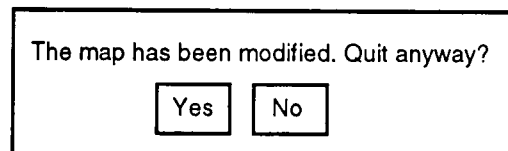


[NoticeYN_FileOverwrite]

If *Save* is selected, the file is overwritten. If *Cancel* is selected, the *Notice* disappears and the *Save Map* window remains on the screen for the user to enter a different file name. If an error occurs during save, a notice appears detailing the error (either "File could not be opened for write." or "Error writing file."); when dismissed, the *Save* box is put back on the screen to try again.

Once *Save* has successfully been selected, a file is written out with a comment at the top with the filename, current date and time, a note that it was automatically written by the named program, and

Selection of *Quit* terminates the program. If any map objects have been modified or added, then the following notice appears:



[NoticeOK_MapModQuit]

If No is selected, then the program continues operation. Otherwise, if Yes was selected or no map objects were modified or added, then QUIT events are sent via HELIX to all simulation processes that are registered with the program, and the program terminates.

b. The Map Menu

The Map menu consists of the following options:

- Redraw [default]
- <blank line>
- Update HELIX Map
- Clear Map
- Change Map Size...
- New Map Object...

Selection of *Redraw* causes all objects (robots or map objects) in the main window to be redrawn. This also occurs if the window is resized via the window manager.

Selection of *Update HELIX Map* causes the bitmap representation of the map in HELIX memory to be cleared and then rescanned. Once this is complete, a MAP_UPDATED message is sent to all registered clients.

If the user selects *Clear Map* and the map is currently empty, then nothing happens. If, however, the user select Clear and there are objects in the map, then the following notice appears:

Are you sure you want to clear the map?

YesNo

[NoticeYN_ClearMap]

If the user answers Yes, then the map is cleared of all objects but the map size remains the same; the screen is also redrawn to reflect the new blank map. If the user answers No, then there is no change.

The *Change Map Size...* selection causes the following window to appear:

—○Map Size

Width: <fl%.2f>Length: <fl%.2f>

Scale: __<int>__☒ ☒

Change

[PopupMapSize]

If Change is then selected, the values of the width and length fields are checked. If they are less than 1.0 or not legal numbers, then a notice box appears with an OK button and one of the following messages: "Width and Length must be at least 1 m." or "Width and Length must be floating point numbers." Otherwise, the map size is changed, and the map is redrawn with objects and robots to scale.

The *New Map Object...* selection causes the following window to appear:

—○New Map Object

Object:

Box

Cylinder

Polyline

New

[PopupNewMapObj]

The three top items are an exclusive choice list, with Box being the default choice when the box appears for the first time. If the pin is pressed or the box ignored, then nothing happens. If the New button is pressed, the New Map Object box disappears and one of the following windows appears, depending on the selection made:

☐
Map Object: Box

Map File: <string>

Position x: <fl %2f> y: <fl %2f>

Width: <fl%.2f> Length: <fl%.2f> Height: <fl%.2f>

☒ visible

[PopupMapObjBox]

☐
Map Object: Cylinder

Map File: <string>

Position x: <fl%.2f> y: <fl%.2f>

Radius: <fl%.2f> Height: <fl%.2f>

☒ visible

[PopupMapObjCylinder]

☐
Map Object: Polyline

Map File: <string>

Position x: <fl%.2f> y: <fl%.2f>

Height: <fl%.2f>

Point List:

_<fl%.2f>,<fl%.2f>,...

☒ visible

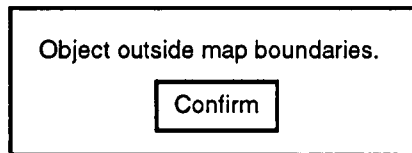
[PopupMapObjPolyline]

For a new object, Map File: has "<new>" in it, and all other values are 0.0. If the user pushes the pin, the new item is not created. If Change or Delete is pressed and Add has not yet been pressed, then the following notice appears:

The object has not been added.

[NoticeOK_ObjNotAdded]

If Add is pressed and any of the Length, Width, or Height parameters are zero or less, then a notice with an OK button and the following message appears: "Length/Width/Height must be greater than zero". If the object has been placed so all or part of it is outside the current map size, then the following notice appears:



[NoticeOK_ObjOutside]

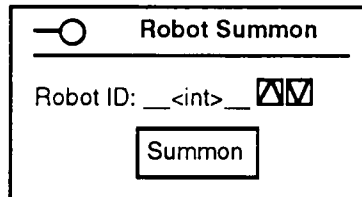
Otherwise, the item is added to the Map, and the map will be redrawn to show the object.

c. The Robots Menu

The Robots menu consists of the following options:

Summon... [default]
 <blank line>
 Start all
 Stop all
 Quit all

Selection of *Summon...* causes the following box to appear on the screen:



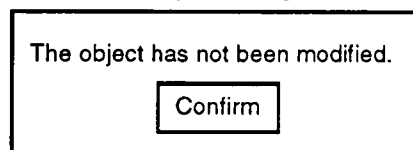
[PopupRobotSummon]

The box can be dismissed via the Summon button or the pin. If the summon button is pressed and a valid robot ID has been entered (range of 1...1000), then the Robot Control box (below) appears. If the ID is invalid, then a notice appears with the following message: "Invalid I.D.". The up/down arrows next to the field allow the number to be incremented and decremented, but not below 1. The maximum is 1000. Note that this box can be handy if the robot has driven off screen and the mouse actions below cannot be used.

Selection of *Start All* sends START events via HELIX to all simulation processes registered with the program. *Stop All* sends STOP events similarly, and *Quit All* sends QUIT events similarly.

4. Mouse Actions

If the left mouse button is clicked while the pointer is inside a map object, a box from one of the three Map Object boxes (above) appears, depending on the type of object. Changes can be made to the object. If no changes are made, then pressing the Add or Change button gives the following message:



[NoticeOK_ObjNotMod]

Otherwise, the Add acts as given above, and replaces the Map File string with <added>. Change replaces the current object with the given modifications, and replaces the Map File field with <changed>. If Delete is pressed, then the object is removed from the map. If any of Add, Change, or Delete are successfully done, then the Map is redrawn to show the object change.

If the middle mouse button is clicked inside an object, the Map Object box appears and then the object tracks the mouse moving around the screen. When the mouse button is released, the equivalent of the Change button is done, i.e. the map is updated and redisplayed.

If the left mouse button is clicked while the pointer is inside a robot (or within some reasonable bounding box, since robots may be round or rectangular but not oriented with the XY axes), the following window appears:

○
Robot Control

Robot ID: ☒ ☒
 Robot Type:

Position x: y: az: Speed:

Status:

Single Update

Cont. Update:

Start

Stop

Send Change

Events ↓

Add-Ons ↓

[PopupRobotControl]

This box provides not only basic status information on the robot retrieved from HELIX memory. It can be dismissed as the with the map object box. This box also features two pull-down menus and a special row of buttons, discussed below. If the middle button is pressed inside a robot, then the window also appears. In this case, however, the interface sends a REQUEST_PLACEMENT message to the robot. If the robot returns PLACE_OK, then the robot moves around the screen and follows the mouse for location and az (as with initial placement); once the placement is finished, the new position is set in HELIX memory and PLACE_SET is sent to the robot. If the robot returns PLACE_REFUSE, then the following notice appears:

The robot has refused the update.

Confirm

[NoticeOK_UpdateRefused]

The Events menu has the following items:

Start
Stop
Ping [default]
Quit

Each of these send a corresponding HELIX event to the robot.

The Add-Ons menu is initially empty, but can be added to via events sent from add-on programs.

The Single Update button reads the latest data from the robot in the HELIX memory and displays that in the box. If the Cont Update/Start button is selected, then the display is periodically updated from HELIX memory, not necessarily as fast as other graphics updates on the screen. When the Cont Update/Stop button is selected, the screen is not updated unless Single Update is chosen. If the Send Change button is pressed, the interface sends a REQUEST_PLACEMENT message to the robot. If the robot returns PLACE_OK, the new position is set in HELIX memory and PLACE_SET is sent to the robot. If the robot returns PLACE_REFUSE, then the notice shown above appears.

If the Robot ID value is changed, then the screen is updated for that robot. If Cont Update/Start button is selected, then continuous updates are displayed for the newly selected robot. If the Robot ID value is not valid for a registered robot, then <invalid ID> is displayed in the Status field.

5. X Events/Window Manager Actions

There are some events that have to be monitored by the program that are a result of user actions but come from the window manager rather than the program's direct interactions. These can usually be monitored via call-backs and could therefore be considered stimuli rather than conditions; this will depend on the specifiers preference. See "System Interfaces" below for information on more system conditions.

These conditions [Heller92] and their cooresponding actions:

```
X:Destroy(status) where status =  
    DESTROY_CHECKING -->  
        if map needs saving, ask if ok to quit; if not, veto destroy; otherwise send QUIT  
        messages to all registered simulation components and allow destroy.  
    DESTROY_SAVE_YOURSELF -->  
        ignore  
    DESTROY_CLEANUP -->  
        prepare for death; should clean up memory  
    DESTROY_PROCESS_DEATH -->  
        prepare for death; no need to clean up memory  
  
X:Repaint -->    redraw map and robot objects
```

B. HELIX INTERFACE AND INTERACTIONS

1. HELIX Events

Once the program has been invoked and the initial screen drawn, the program can respond to the basic set of HELIX events. These are NULL_EVENT, QUIT, EPING, ACK, QUIET, START, and STOP. The actions for these are the following:

```
NULL --> no response  
QUIT --> terminate program; send QUIT to registered simulation processes  
EPING --> return ACK event to sending process  
QUIET --> no response  
START --> no response  
STOP --> no response  
ACK --> no response
```

In addition, a set of events has been defined especially for interfacing with simulation components. These are: ROBOT_STARTUP, ROBOT_SHUTDOWN, PLACE_REQUEST, PLACE_OK, PLACE_REFUSE, PLACE_SET. ROBOT_STARTUP is sent by an initializing simulation component to register itself with the interface. ROBOT_SHUTDOWN is sent by a robot when it is shutting down to let the interface know that it should dispose of the robot. Robots send this when they have received a QUIT, usually from the interface.

When ROBOT_STARTUP is received, the interface will have to register the robot that the message was received from. If the robot wishes to place the robot in the environment, it also sends a PLACE_REQUEST message. With this message, the interface goes into a special mode for user interaction. The robot image is drawn on the screen at the mouse position, and follows the mouse around. If the middle mouse button is pressed, the robot rotates its az value counter-clockwise. If the right mouse button is pressed, the robot rotates its az value clockwise. If the left mouse button is pressed, the robot is placed at that point, and a PLACE_SET message is sent back to the robot simulation component. Note that there is no way to abort the robot placement sequence. HELIX events received during this sequence are queued. If the PLACE_REQUEST message is not received, it is assumed that the robot simulator has selected the position of the robot before sending ROBOT_STARTUP, and the robot is drawn at the coordinates given in the HELIX shared memory.

The other HELIX interface is with Add-On components. At this time, this feature will not be included.

The following table illustrates the HELIX communications that takes place under certain specific activities of the simulations and the interface:

Activity	Interface	Simulation Component (Robot)
robot startup	register robot draw on screen	send <-- ROBOT_STARTUP
robot placement	locate robot via mouse put location in HELIX memory send <-- PLACE_SET	send <-- PLACE_REQUEST
interface shutdown	send QUIT -->	finish simulation send <-- ROBOT_SHUTDOWN
simulator shutdown	deregister robot, remove from screen	send <-- ROBOT_SHUTDOWN
map update	update map in shared memory, send MAP_UPDATED -->	
robot position adjust (via interface)	send PLACE_REQUEST --> if PLACE_OK received, locate robot via mouse put location in HELIX memory send PLACE_SET --> else if PLACE_REFUSE, notify user of refusal	send PLACE_OK if placement ok; send PLACE_REFUSE if not
robot position read	read robot position from HELIX memory	
robot position update		write robot position to HELIX memory

2. HELIX Shared Memory

a. Robot/Simulator Interface

This section discusses the PostIts in shared memory which will be required. The interface will have to determine the current robot status and position. Therefore the following PostIt will be needed for each robot:

```
struct robot_comm {
    float x;           /* x in meters */
    float y;           /* y in meters */
    float az;          /* az in radians */
    float speed;        /* speed in m/sec */
    char status[80];    /* string holding robot status info */
    int time;           /* time stamp to indicate update */
    short type;         /* robot type */
    short global_ID;    /* ID of the robot vs. all others */
    short intype_ID;    /* number of this robot among those
                        with the same type */
    short flags;        /* flags
                        bit0 = 1 if real robot, 0 if sim */
};
```

Also, the map of visible and invisible objects has to be placed in memory. The `vis_objs_map` contains only the visible objects; `all_objs_map` contains both visible and invisible objects. The maps are currently fixed size, and sized to fit into 32k, based on an OS-9 HELIX limitation. If a map is smaller than 12.8m x 12.8m, then the space beyond that size is considered empty and is cleared by the interface. If a map is larger than 12.8m x 12.8m, it will be cropped at the 12.8m x 12.8m border for HELIX representation. This may be fixed in a future version.

```
struct vis_objs_map {
    char grid[128][128][32];
}

struct all_objs_map {
    char grid[128][128][32];
}
```

b. Add-On Interface

A PostIt will be needed for the interface to communicate certain information with Add-Ons. For now, this information is nil.

```
struct add_on_interface {
}
```

C. SUPPORTING SPECIFICATIONS

1. Map Format

The map file consists of lines of interpretable data. Each line is considered a separate entity. Each map line can be nil (blank line), start with a # (comment), or have a map command, one of "map", "cyl[inder]", or "box". The <nws> (non-white space) after the command means that the rest of the command name is ignored if the first three letters are matched, i.e. cylinder, cyl, or cylfoo are all legal for cylinder. The <vis> parameter, which is optional, indicates whether an object is visible (i.e. a priori known to the robot) or invisible (only can be found via sensors). The default condition is visible. The <height> parameter, if not present, defaults to 2m.

Maps may be overlayed with one another, i.e. multiple map files may be loaded to make a single complex map in the simulation memory. If, however, separate maps are loaded and have different map sizes, the largest map size is assumed. The map size defaults to 12.8m x 12.8m and to a scale of 40; the default is used if no map command is encountered.

The following is a grammar for interpreting a map file.

```
Map := [<Line><CR>]*
Line :=
    nil
    "#"<anychar>*|
    "map"<nws><ws><map-width><ws><map-length>|
    "map"<nws><ws><map-width><ws><map-length><ws><map-scale>|
    "cyl"<nws><ws><vis><locx><ws><locy><ws><radius>|
    "cyl"<nws><ws><vis><locx><ws><locy><ws><radius><ws><height>|
    "box"<nws><ws><vis><locx><ws><locy><ws><width><ws><length>|
    "box"<nws><ws><vis><locx><ws><locy><ws><width><ws><length><ws><height>|
    "poly"<nws><ws><line-list>|
    "poly"<nws><ws><height><ws><line-list>
```

line-list := <locx0>,<ws><locy0><middle><locx0>,<ws><locy0>
middle := [<locxi>,<ws><locyi><ws>]* | <ws>

NOTE that the first coordinate must be repeated to close the polyline

ws := <tab>|<space>

nws := any character not <ws> or <CR> (including nil)

vis := "vis"<nws><ws>|"invis"<nws><ws>|<ws>

nil := empty string

2. System Interfaces

a. File System (FS)

The file system will present several conditions to the system which will have to be dealt with. These are already dealt with in the requirements above. Errors include: FS:FileExists, FS:FileDoesNotExist, FS:WriteError, FS:ReadError.

b. Memory Management (MM)

This system is not dealt with in the requirements above. If a memory allocation (object constructor, malloc, etc) returns an error, program should put up a notice that memory is low. A QUIT event is send to all registered simulation components. The program then checks to see if a save is needed; if so, brings up the save box to save the current map. After a save is complete, the program quits automatically. All menu items except Quit and Save are disabled during this procedure. If a second memory error occurs during this shutdown procedure, the program terminates immediately.

D. SUMMARY OF GUI ELEMENTS

Windows/Popups/Notices

WinMap

PopupLoadMap
 PopupSaveMap
 PopupMapSize
 PopupNewMapObj
 PopupMapObjBox
 PopupMapObjCylinder
 PopupMapObjPolyline
 PopupRobotSummon
 PopupRobotControl

NoticeOK

NoticeYN

Menus

Window	Menu	Item
MapWin	File	Load...
		Save...
		Quit
	Map	Redraw [default]
PopupRobotControl	Robots	Update HELIX Map
		Clear Map
		Change Map Size...
		New Map Object...
		Summon... [default]
	Events	Start All
		Stop All
		Quit All
		Start
		Stop
	Add-Ons	Ping [default]
		Quit
		<none>

Misc. Stimuli

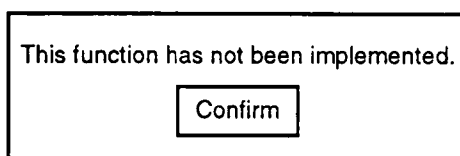
XV/Destroy

XV/Repaint

III. Incremental Development Plan

This section describes the incremental development plan. This plan roughly states which features will be included into each increment as the software is developed. Each increment will be specified, coded, verified, and tested individually. Each increment also must a functional, runnable program or group of programs.

If functions have buttons or menus in an early increment but are not yet implemented, the following notice appears if they are selected:



[NoticeOK_FuncNotImp]

Increment 1

Initial window with menus and all menu items present. All menu items are stubs, except Redraw and PopUpMapSize will be available.

Increment 2

Map loading, saving, clearing, and drawing on screen available; only box and cylinder objects allowed (NOTE: polylines not included until increment 7).

Increment 3

On-screen editing of map objects allowed via mouse clicks. New map objects may be created. Visible and invisible objects, map size information in file.

Increment 4

Initial robot/HELIX interface. Basic event interface + robot_status interface as defined in this document. Object map from interface not yet available.

Increment 5

Addition of robot control via mouse clicks on screen. Robot control box and submenus added. Robot Summon available.

Increment 6

HELIX shared memory object map supported by interface.

Increment 7

Polyline objects added.

Subappendix A: The XView Toolkit

A Note on XView Interface Elements

Figure 3 shows some of the graphical elements used in the sample screen displays below [Sun89].

Control areas, as shown in Fig 3, are required for certain interface elements such as buttons, menus, slides, text input, etc. Another type of area on a window is a Pane. This area is typically a text or graphics area (the latter being called a Canvas).

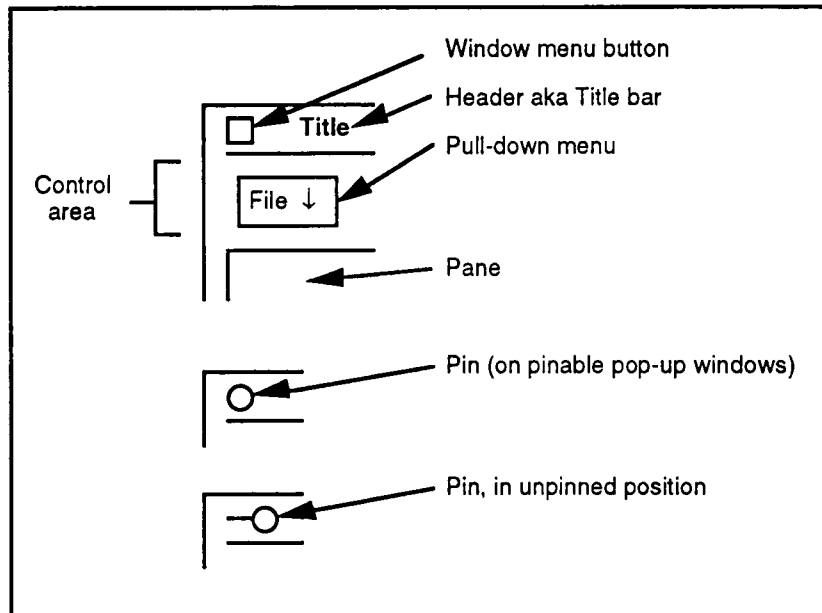


Figure 3: XView Interface Elements

Pins appear on certain windows which may be designed to stay on the window if the user chooses. If a window comes up with the pin already stuck in, then the window will stay in place unless the user explicitly dismisses it or pushes on the pin. If the pin comes up unpinned, then the window will disappear after it is used unless the user pins it in place. If the window is made to disappear by unpinning it, this is equivalent to cancelling any function the window was to perform. Note that actions using the pins do not send a specific stimulus to the program. Thus, if a box is cancelled by pushing its pin, the program does not note that the box has disappeared, but will never receive any other stimuli from the box.

Note that using pin-able pop-up windows allows for non-modal operations, i.e. it is possible to bring up these windows, but then continue working with other windows. There are also modal windows, that must be acted upon before other work can be done with the program. In XView, these are called Notices, and they are used for some functions in the program below.

Menus are implemented via buttons that, when clicked on via the right mouse button, present a pull-down menu. If the left mouse button is pressed, a default selection (if set by the programmer) is highlighted automatically.

Why XView?

The graphical user interface will be built using the XView toolkit. This toolkit has been chosen because (a) we have an interface builder for this toolkit and (b) it is the basis for the window tools we use in our laboratory (Sun's OpenWindows). Writing programs for basic X windows is more universal, but also much

harder. Using a toolkit such as XView or Motif will save a considerable amount of time. XView was selected over Motif because it is better supported in our laboratory, and I have been told that it is easier and more logical to program than Motif. One final note: XView libraries are available free, which should allow this to run on any X windows-based system.

Subappendix B: Glossary

Add-ons—new interface elements added to provide information from a specific simulation.

Component—a process or group of processes that represent a significant part of the simulation—i.e. a robot or the interface.

Cooperation Process—the process in a component that is responsible for communicating with the other components in the system, including other simulation components and the interface.

Element—a part of a component, i.e. a process that is part of a single robot simulator, or a process that is part of the interface.

GUIDE—A Sun-based tool for developing XView interfaces. GUIDE stands for Graphical User Interface Development Tool.

HELIX—a system for communications between processes, including message passing (events in HELIX terminology) and shared memory (PostIts in HELIX terminology). N-HELIX is an extension to HELIX that supports a hierarchy of HELIX networks.

Interface—refers to the component of the simulation that presents the graphical user interface to the user. This component is also called the “Master Simulator” since it controls the actions of the other components

Registered Process/Robot/Simulator—a robot simulator is “registered” with the interface if it has sent a ROBOT_STARTUP event to the interface, indicating that its position should be tracked.

Robot—a component of the simulation system, possibly consisting of multiple processes, and either controlling a real robot or a simulated robot.

XView—a Sun-developed X-based graphical user interface library which is comparable to Motif. The system will use this library as it is available on all our Suns.

Subappendix C: References

- [Asama92] Asama, H., Conversations with Dr. Asama and lab personnel during visit to RIKEN, July 28, 1992.
- [Gowdy91] Gowdy, J. and C. Thorpe, "The EDDIE System: An Architectural Toolkit for Mobile Robots," Robotics Institute, Carnegie Mellon, February 8, 1991.
- [Habib92] Habib, M.K., *et al.*, "Simulation Environment for An Autonomous and Decentralized Multi-Agent Robotic System," *Proceedings of the 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Raleigh, NC, July 7-10, 1992.
- [Heller92] Heller, D., *XView Programming Manual*. 3rd ed. The X Window System, Vol. 7. 1992, O'Reilly & Associates.
- [Jones92a] Jones, J.P., A.L. Bangs, and P.L. Butler, "A System for Simulating Shared Memory in Heterogeneous Distributed-Memory Networks with Specialization for Robotics Applications," *Proceedings of the IEEE International Conference on Robotics and Automation*, Nice, France, May 12-14, 1992.
- [Jones92b] Jones, J.P., *et al.*, "Hetero Helix: Synchronous and asynchronous control systems in heterogeneous distributed networks," *Robotics and Autonomous Systems*, 1992. 10(4).
- [Kimoto92a] Kimoto, K. and S. Yuta, "A Simulator for Programming the Behavior of an Autonomous Sensor-Based Mobile Robot," *Proceedings of the 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Raleigh, NC, July 7-10, 1992.
- [Kimoto92b] Kimoto, K., Personal Communication, August 12, 1992.
- [Mont90] Montgomery, T.A. and E.H. Durfee, "Using MICE to study intelligent dynamic coordination," *Proceedings of the Second Computer Society International Conference on Tools for Artificial Intelligence*, November 1990.
- [Parker92b] Parker, L.E., Personal Communication, October 22, 1992.
- [Parker92c] Parker, L.E., "Adaptive Action Selection for Cooperative Agent Teams," *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, MIT Press, November 1992.
- [Sun89] Sun Microsystems Inc., *Open Look™ Graphical User Interface Functional Specification*. 1989, Addison-Wesley.
- [Sun91] Sun Microsystems Inc., *xview(3) man page*, 1991.
- [Torrance92] Torrance, M.C., "The Case for a Realistic Mobile Robot Simulator," *AAAI Symposium on Applications of Artificial Intelligence to Real-World Autonomous Mobile Robots*, Cambridge, MA.
- [Watanabe93] Watanabe, Y., Personal Communication, Jan 28, 1993.

Appendix C: The First Increment Specification

WestWorld

Increment 1 Specification

I. Top Level Black Box

define_BB void WestWorld

input

Invocation

XCB¹:Menu/File/Load

XCB:Menu/File/Save

XCB:Menu/File/Quit

XCB:Menu/Map/Redraw

XCB:Menu/Map/UpdateHelix

XCB:Menu/Map/Clear

XCB:Menu/Map/ChangeSize

XCB:Menu/Map/NewObj

XCB:Menu/Robots/Summon

XCB:Menu/Robots/Start

XCB:Menu/Robots/Stop

XCB:Menu/Robots/Quit

XCB:Button/MapSizeChange

XCB:XV/Destroy

XCB:XV/Repaint

output

window WinMap

popup PopupMapSize

notice NoticeOK

transition

Invocation -->

_WinMapInit()

XCB:Menu/File/Load -->

_Unimplemented(WinMap frame created at Invocation);

XCB:Menu/File/Save -->

_Unimplemented(WinMap frame created at Invocation);

XCB:Menu/File/Quit -->

xv_destroy_safe(WinMap frame created at Invocation)

XCB:Menu/Map/Redraw -->

_Repaint()

XCB:Menu/Map/UpdateHelix -->

_Unimplemented(WinMap frame created at Invocation);

XCB:Menu/Map/Clear -->

_Unimplemented(WinMap frame created at Invocation);

XCB:Menu/Map/ChangeSize -->

_PopupMapSizeShow()

XCB:Menu/Map/NewObj -->

_Unimplemented(WinMap frame created at Invocation);

XCB:Menu/Robots/Summon -->

_Unimplemented(WinMap frame created at Invocation);

XCB:Menu/Robots/Start -->

_Unimplemented(WinMap frame created at Invocation);

¹XCB = X Call Back

```

XCB:Menu/Robots/Stop -->
    _Unimplemented(WinMap frame created at Invocation);
XCB:Menu/Robots/Quit -->
    _Unimplemented(WinMap frame created at Invocation);
XCB:Button/MapSizeChange -->
    _PopupMapSizeChange()
XCB:XV/Destroy(client, status) -->
    _Destroy(client, status)
XCB:XV/Repaint -->
    _Repaint()

```

end_BB

Black Box Specification Functions:

```

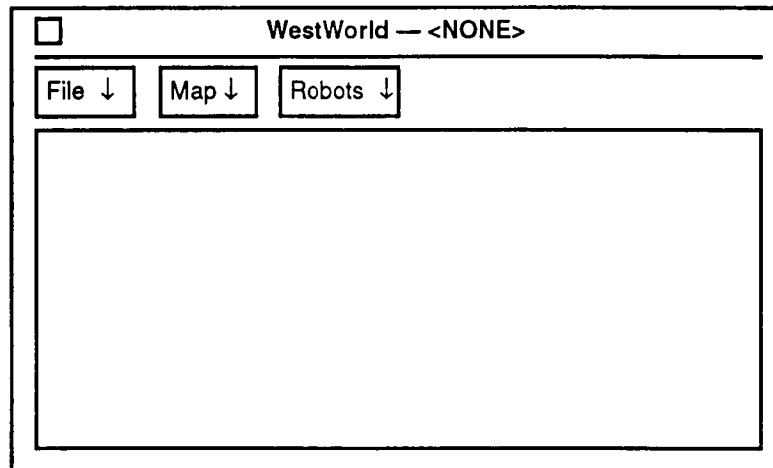
[ _WinMapInit() ] ≡
[

```

```

    display WinMap on screen, with canvas exactly encompassing default map size
    with title "WestWorld -- <None>"
    with menus as follows:
        File: Load..., Save..., Quit
        Map: Redraw <default>, <blank>, Update HELIX Map,
            Clear Map, Change Map Size..., New Map Object...
        Robots: Summon... <default>, <blank>, Start All,
            Stop All, Quit All
    with border fitting map size (_Repaint)

```



```

]

```

```

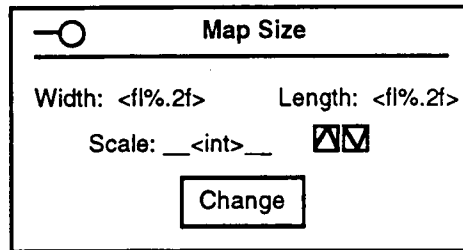
[ _Destroy(Xv_opaque client, Destroy_status status) ] ≡
[
    status = DESTROY_CHECKING --> NOTIFY_DONE
    | status = DESTROY_SAVE_YOURSELF --> NOTIFY_DONE
    | status = DESTROY_CLEANUP --> notify_next_destroy_func(client, status)
    | status = DESTROY_PROCESS_DEATH --> NOTIFY_DONE
]

```

```

[ _PopupMapSizeShow() ] ≡
[
    display PopupMapSize with map width (using %.2f format) of default of 12 m or last width
    set by successful XCB:Button/MapSizeChange and length (using %.2f format) of 12m or last
    height set by XCB:Button/MapSizeChange and scale of default 40 or last scale set by
    XCB:Button/MapSizeChange

```



A dialog box titled "Map Size" with a close button (a circle with a horizontal line) in the top-left corner. The dialog contains three input fields: "Width: <f%.2f>", "Length: <f%.2f>", and "Scale: __<int>__". To the right of the "Scale" field are two checked checkboxes. At the bottom center is a "Change" button.

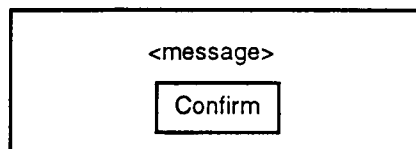
]

```
[ _PopupMapSizeChange() ] ≡
    [ (atof2(entered width) < 1 | atof(entered length) < 1) -->
        NoticeOK(PopupMapSize frame,
            "Width and Length must be at least 1.0m.")
    | (entered scale < 1) | (entered scale > 100)
        NoticeOK(PopupMapSize frame,
            "Scale must be in the range of 1 to 100.")
    | (PopupMapSize pushpin is in) -->
        redisplay values in PopupMapSize;
        _Repaint()
    | true -->
        _Repaint()
    ]

[ _Repaint() ] ≡
    [ clear WinMap paint window created at Invocation and
        draw map border using X Display+Window params for paint window
        given width, length, scale from
        default or XCB:PopupMapSizeChange
    ]
```

```
[ _Unimplemented(Xv_opaque owner) ] ≡
    [ _NoticeOK(owner, "This function has not been implemented.") ]
```

```
[ _NoticeOK(Xv_opaque owner, char *message) ] ≡
    [ display notice for owner
        with Confirm button and given message string:
```



A dialog box containing a message field labeled "<message>" and a "Confirm" button at the bottom center.

]

²atof() is a C function which converts a string to a floating point number, ignoring any alphanumeric characters.

II. Class Design and Class BB Specifications

(1) Choose candidate objects

The first candidates for objects/classes are those that represent an interactive window under X; this is the way the code is automatically generated by the Devguide tool. Therefore, there needs to be an object for the main window and its menus (WinMap) as well as the "Change Map Size" popup (PopupMapSize). There does not need to be one for the Notify boxes since they do not have any notion of complex interaction or permanence. In addition, there needs to be an object to hold the map data and associated functions (Map). A class should be created to group all the miscellaneous functions, such as Notice_OK (Utils). The final object is that which controls all the others, or at least initiates their actions via the main() function (Main).

(2) Assign top-level stimuli to objects

Main

Invocation

Map

<none>

PopupMapSize

XCB:Button/MapSizeChange

WinMap

XCB:Menu/File/Load

XCB:Menu/File/Save

XCB:Menu/File/Quit

XCB:Menu/Map/Redraw

XCB:Menu/Map/UpdateHelix

XCB:Menu/Map/Clear

XCB:Menu/Map/ChangeSize

XCB:Menu/Map/NewObj

XCB:Menu/Robots/Summon

XCB:Menu/Robots/Start

XCB:Menu/Robots/Stop

XCB:Menu/Robots/Quit

XCB:XV/Destroy

XCB:XV/Repaint

Utils

<none>

(3) Identify inter-class stimuli

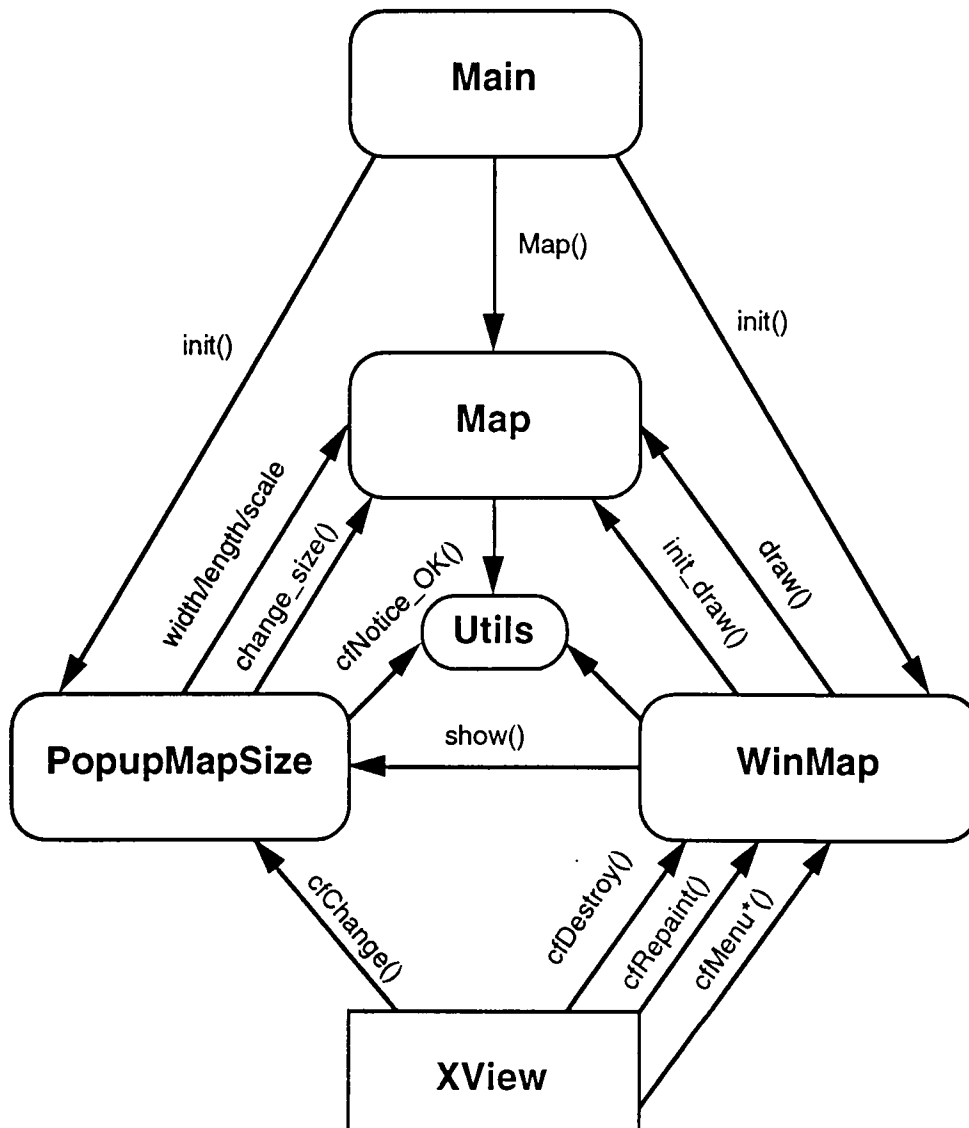
Main, via the main() function, will have to initialize/create all the other objects, through either init calls or constructors.

Map will draw the map info in the paint window of WinMap, based on window information passed from WinMap. It will assume a default map size until a map size change is sent from another object. It should have the basic map parameters publicly available.

PopupMapSize will draw the Change Map Size popup and handle the button callback for that popup. It will have to be able to display the popup on command when the appropriate menu item is selected via WinMap, and it will have to pass the change size parameters to Map when the Change button is pressed.

WinMap will be responsible for drawing the main window. It will accept all menu callbacks, but will only handle those directly related to what it controls. Since Map will handle the drawing of the map in a subpane of WinMap, data referencing that subpane will have to be passed to Map, as will the actual draw calls. The menu item that causes the Change Map Size popup to appear will have to be passed to PopupMapSize.

Utils will handle the Notice_OK call.



BB Format Notes:

- (1) *access programs* includes any access to class via function calls; if a return or I/O parameter is involved, this must be handled in the response section of the transition for this access program.
- (2) *output variables* shows variables maintained by the box which are publicly accessible; access to these variables is to be considered a stimuli for the purpose of determining the value of the output
- (3) *output* contains output not handled by access program parameters or return values -- i.e. user interface outputs
- (4) *class access*, *class output variables*, and *class output* provide a similar interface as described above, but for the class functions.

- (5) *input variables* lists external inputs required by this class.
 (6) *external access* lists external function calls required by this class

define_BB Main

```

class access programs
  <Invocation>
  main()

output variables
  Attr_attribute INSTANCE

input variables
  WinMap::frame

external access
  Map::Map()
  void PopupMapSize::init(Xv_opaque owner_frame, Map* pMap)
  void WinMap::init(Xv_opaque owner_frame, Map* pMap,
    PopupMapSize *pPopupMapSize)

transition
  Si = <Invocation> -->
    create Map [invokes Map()], PopupMapSize, WinMap
  Si = main() -->
    call init for PopupMapSize + WinMap object created by Invocation

```

end_BB

define_BB Map

```

access programs
  Map()
  void init_draw(Display *display, Window xid)
  int change_size(Xv_opaque frame, double new_width, double new_length,
    int new_scale)

output variables
  double width
  double length
  double scale

output
  X display window

external access
  void Utils::cfNotice_OK(Xv_opaque owner, char* message)

transition
  Si = Map() --> No response.
  Si = init_draw(display, xid) -->
    draw map border (rectangle) of 12*40 x 12*40, using display + xid parameters.
  (Si = change_size(frame, w, l, s)) ^ ((w < 1) ∨ (l < 1)) -->
    Utils::NoticeOK(frame, "Width and Length must be at least 1.0m.");
    return FALSE value from change_size
  (Si = change_size(frame, w, l, s)) ^ ((s < 1) ∨ (s > 100)) -->
    Utils::NoticeOK(frame, "Scale must be in the range of 1 to 100.")
    return FALSE value from change_size

```

```

(Si = change_size(frame, w, l, s)) ∧ change_valid(w, l, s) -->
    draw map border (rectangle) of new_width*new_scale x new_width*new_scale
    using display + xid parameters from previous init_draw() call;
    return TRUE value from change_size
Si = width ∧ (∃Sj | (j < i) ∧ (Sj = change_size(f,w,l,s) ) ∧ change_valid(w,l,s)) ∧
not(∃Sk | (j < k < i) ∧ (Sk = change_size(f,w',l',s') ) ∧ change_valid(w',l',s')) --> w
Si = width ∧ not(∃Sj | (j < i) ∧ (Sj = change_size(f,w,l,s) ) ∧ change_valid(w,l,s)) --> 12
Si = length ∧ (∃Sj | (j < i) ∧ (Sj = change_size(f,w,l,s) ) ∧ change_valid(w,l,s)) ∧
not(∃Sk | (j < k < i) ∧ (Sk = change_size(f,w',l',s') ) ∧ change_valid(w',l',s')) --> l
Si = length ∧ not(∃Sj | (j < i) ∧ (Sj = change_size(f,w,l,s) ) ∧ change_valid(w,l,s)) --> 12
Si = scale ∧ (∃Sj | (j < i) ∧ (Sj = change_size(f,w,l,s) ) ∧ change_valid(w,l,s)) ∧
not(∃Sk | (j < k < i) ∧ (Sk = change_size(f,w',l',s') ) ∧ change_valid(w',l',s')) --> s
Si = scale ∧ not(∃Sj | (j < i) ∧ (Sj = change_size(f,w,l,s) ) ∧ change_valid(w,l,s)) --> 40

```

end_BB

Spec Function

```

[ change_valid(w,l,s) ] ≡
    [ ((1 ≤ s ≤ 100) ∧ (w ≥ 1) ∧ (l ≥ 1)) ]

```

NOTES:

- (1) assumption is made that init_draw comes before any change_size; no error checking for this
- (2) Map() must be first stimuli, by default, since it is a constructor

define_BB PopupMapSize

access programs

```

void init(Xv_opaque owner_frame, Map* pMap)
void show()
void change(Panel_item)

```

output

popup window

class access programs

```

static void cfChange(Panel_item, Event)

```

input variables

```

Attr_attribute INSTANCE;
xv_get variables FRAME_CMD_PUSHPIN_IN, XV_KEY_DATA, entered_width,
    entered_length, entered_scale

```

external access

```

Map::change_size()
Map::width, Map::length, Map::scale

```

transition

```

Si = init(o, p) --> no response.
Si = show() -->
    display popup screen with owner o, with values in width/length/scale fields
    from p->width, p->length, p->scale, where (∃Sj | (j < i) ∧ (Sj = init(o,p)) ∧
    not(∃Sk | (j < k < i) ∧ (Sk = init(o,p))))

Si = change(item) -->
    given pointer to popup input fields for width/length/scale and popup frame "f"
    created by init (∃Sj | (j < i) ∧ (Sj = init(o,p))), call p->change_size(f, entered
    width, entered length, entered scale); if change_size returns 1 and xv_get

```

parameter FRAME_CMD_PUSHPIN_IN from f is 1, then call show(); if change_size() returns 0, send an error to XView via item to hold the popup on the screen.

$S_i = \text{cfChange}(\text{item}, \text{ev}) \rightarrow$
 call $\text{PopupMapSize} * p \rightarrow \text{change}(\text{item})$ where $p = \text{xv_get}(\text{item}, \text{XV_KEY_DATA}, \text{INSTANCE})$

end_BB

NOTES:

(1) assumption is made that init() comes before any other calls

define_BB WinMap

access programs

init(Xv_opaque owner, Map* pMap, PopupMapSize* pPopupMapSize)
 void unimplemented()
 void quit()

output variables

Xv_opaque frame

output

XView main window

class access programs

static Menu_item cfMenuFileQuit(Menu_item, Menu_generate)
 static Menu_item cfMenuMapRedraw(Menu_item, Menu_generate)
 static Menu_item cfMenuMapChangeSize(Menu_item, Menu_generate)
 static Menu_item cfMenuUnimplemented(Menu_item, Menu_generate)
 static void cfRepaint(Canvas, Xv_window, Display, Window, Xv_xrectlist)
 static void cfDestroy(Xv_opaque, Destroy_status)

class output variables

Notify_value notify_value

input variables

Attr_attribute INSTANCE;
 xv_get variable XV_KEY_DATA

external access

PopupMapSize::show()
 Map::init_draw()
 Utils::cfNotice_OK()

transition

$S_i = \text{init}(o, p1, p2) \rightarrow$
 create WinMap window with owner o, call $p1 \rightarrow \text{init_draw}$ with display and xid
 $S_i = \text{unimplemented}() \rightarrow$
 Utils::cfNotice_OK(f, "This function has not been implemented.")
 where f is frame created from S_j , where $(\exists S_j \mid (j < i) \wedge (S_j = \text{init}(o, p)))$
 $S_i = \text{quit}() \rightarrow$
 call $\text{xv_destroy_safe}(\text{frame created from } S_j)$, where $(\exists S_j \mid (j < i) \wedge (S_j = \text{init}(o, p)))$
 $S_i = \text{cfMenuFileQuit}(\text{item}, \text{op}) \rightarrow$

```

        call WinMap* p->quit() where p = xv_get(item, XV_KEY_DATA,
        INSTANCE)
    Si = cfMenuMapRedraw(item, op) -->
        call Map* p->draw() where p = xv_get(item, XV_KEY_DATA, INSTANCE)
    Si = cfMenuMapChangeSize(item, op) -->
        call PopupMapSize* p->show() where p = xv_get(item, XV_KEY_DATA,
        INSTANCE)
    Si = cfMenuUnimplemented(item, op) -->
        call WinMap* p->unimplemented() where p = xv_get(item, XV_KEY_DATA,
        INSTANCE)
    Si = cfRepaint(c, pw, d, w, x) -->
        call Map* p->draw() where p = xv_get(pw, XV_KEY_DATA, INSTANCE)
    Si = cfDestroy(client, status) -->
        _Destroy(client, status)

```

end_BB

NOTES:

(1) assumption is made that init() comes before any other calls

define_BB Utils

class access programs

```
static void cfNotice_OK(Xv_opaque owner, char *message)
```

transition

```
cfNotice_OK(o, m) -->
```

display XView notice with owner o, message, and Confirm button; wait until
Confirm is pressed.

end_BB

III. TAM Specifications for Classes

CLASS: MAIN

TYPE IMPLEMENTED: <Main>

(1) SYNTAX

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4
<Invocation>					
main()	<void>	<int> argc	<char *> argv		

CLASS OUTPUT VARIABLES

Variable Name	Type	Access
INSTANCE	<Attr_attribute>	publicly accessible

INPUT VARIABLES

Variable Name	Type	Access
WinMap::frame	<Xv_opaque>	direct

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3
Map::Map()	(constructor)			
PopupMapSize::init()	<void>	<Xv_opaque>	<Map*>	
WinMap::init()	<void>	<Xv_opaque>	<Map*>	<PopupMapSize*>

(2) CANONICAL TRACES

$$\text{canonical}(T_C) \leftrightarrow (T_C = \text{<Invocation>}) \vee (T_C = \text{main}())$$

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for <Invocation> and main().

$T_C.\text{<Invocation>} \equiv \text{<Invocation>};$

ADD-TO-TRACE(T_m , Map()) where T_m is trace for Map object created by <Invocation>

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- no partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by LHC.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- <Invocation> is canonical.

Consistency (3): All RHC values are unique:

- One value.

$T_C.main() \equiv$

conditions	equivalences
$T_C = \langle \text{Invocation} \rangle$	main(); ADD-TO-TRACE(T_{wm} , init(NULL, p_m , p_{pms})); ADD-TO-TRACE(T_{pms} , init($p_{wm} \rightarrow \text{frame}$, p_m)); where T_{wm} is trace for WinMap object created by main() and p_{wm} is pointer to that object, T_{pms} is a trace for PopupMapSize object created by main() and p_{pms} is pointer to that object, and p_m is pointer to Map object created by $\langle \text{Invocation} \rangle$
else	%main already called%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- event defined by LHC: traces and pointers used in RHC are specified by event in T_C [$\langle \text{Invocation} \rangle$] or the current event [main()]

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- main() is canonical.

Consistency (3): All RHC values are unique:

- One value, one error.

(4) VALUES

OUTPUT VALUES

$V[\text{INSTANCE}](T) =$

conditions	values
$T = \langle \text{Invocation} \rangle$	%undefined%
$T = \text{main}()$	xv_unique_key()

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Since T is canonical, the conditions partition the canonical trace and therefore give a full partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- N/A

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- One value, one error.

RETURN VALUES

$\langle \text{none} \rangle$

CLASS: MAP

TYPE IMPLEMENTED: <Map>

(1) SYNTAX

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4
Map ³	(constructor)				
init_draw	<void>	<Display *> display	<Window> xid		
draw	<void>				
change_size	<int>	<Xv_opaque> frame	<double> new_width	<double> new_length	<int> new_scale

OUTPUT VARIABLES

Variable Name	Type	Access
width	<double>	public
length	<double>	public
scale	<int>	public
change_ok	<int>	function return
(output screen)	(X display window)	N/A

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4
Utils::NoticeOK	<void>	<Xv_opaque> owner	<char *> message		

(2) CANONICAL TRACES

$$\begin{aligned} & \text{canonical}(T) \leftrightarrow \\ & (T = \text{Map}()) \vee \\ & (T = \text{Map}().\text{init_draw}(d, xw, wf).[\text{change_size}(f, w, l, s)]_{i=0}^1 [\text{change_size}(f', w', l', s')] \wedge \\ & \text{bad_values}(w', l', s')]_{i=0}^1) \end{aligned}$$

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

AUXILIARY FUNCTIONS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4	Arg#5
bad_values	<boolean>	<double> new_width	<double> new_length	<int> new_scale		
parse	<boolean>	<trace>	<trace>	<trace>	<trace>	

³Map() is a constructor, and therefore will automatically be called anytime an instance of the class is created. Map() defines default values for the values width, length, and scale until redefined by change_size. Map() cannot be called explicitly.

bad_values(w,l,s) =

conditions	equivalences
$(w < 1) \vee (l > 1) \vee (s < 1) \vee (s > 100)$	true
else	false

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

parse(S,S1,S2,S3) =

conditions	equivalences
$(S = S1.S2.S3) \wedge (S1 = \text{Map}[\text{init_draw}(d,xw)]_{i=0}^1) \wedge (S2 = [\text{change_size}(f,w,l,s) \wedge \text{not}(\text{bad_values}(w,l,s))]_{i=0}^1) \wedge (S3 = [\text{change_size}(f,w,l,s') \wedge \text{bad_values}(w,l,s')]_{i=0}^1)$	true
else	false

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for access programs Map, init_draw, draw, and change_size.

T.Map() $\equiv$ Map()

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- No partitioning of domain, therefore complete

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- Map() is a canonical trace

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

T.init_draw(d,xw) $\equiv$ Map().init_draw(d,xw).C.CE, where parse(T, I, C, CE)

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- The predicates in RHC are comprised of canonical trace elements from LHC or the stimulus itself, and are therefore all defined.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- The trace given is canonical.

Consistency (3): All RHC values are unique:

- Only one value.

T.draw() =

conditions	equivalences
T = Map()	%uninitialized%
else	T

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T is defined by other side of equivalence.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T is canonical by definition.

Consistency (3): All RHC values are unique:

- One value, one error.

T.change_size(f,w,l,s) =

conditions	equivalences
T = Map()	%uninitialized%
$(w < 1) \vee (l < 1)$	equiv = I.C.change_size(f,w,l,s); ADD-TO-TRACE(T_u , cfNotice_OK(f, "Width and Length must be at least 1.0m.") where parse(T, I, C, CE) and T_u is the class access trace for Utils
$(w \geq 1) \wedge (l \geq 1) \wedge ((s < 1) \vee (s > 100))$	equiv = I.C.change_size(f,w,l,s); ADD-TO-TRACE(T_u , cfNotice_OK(f, "Scale must be in the range of 1 to 100.") where parse(T, I, C, CE) and T_u is the class access trace for Utils
else	I.change_size(f,w,l,s) where parse(T, I, C, CE)

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first case has only constructor, others assume init_draw() in trace; second and third separated by w/l comparisons, else insures full partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- RHC items defined by call and parsed trace.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- The traces shown are all canonical.

Consistency (3): All RHC values are unique:

- Second and third cases are different by the cfNotice_OK calls; third replaces any error present.

(4) VALUES

OUTPUT VALUES

V[width](T) =

conditions	values
$\text{parse}(T, I, C, CE) \wedge C \neq _$	w where $C = \text{change_size}(f, w, l, s)$
else	12

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- first case is defined since $\text{change_size}()$ must be defined if $C \neq _$; second case is constant.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- The first case value may be the same as the constant, but not always, requiring partitioning.

V[length](T) =

conditions	values
$\text{parse}(T, I, C, CE) \wedge C \neq _$	l where $C = \text{change_size}(f, w, l, s)$
else	12

Consistency/Completeness: Same as above.

V[scale](T) =

conditions	values
$\text{parse}(T, I, C, CE) \wedge C \neq _$	s where $C = \text{change_size}(f, w, l, s)$
else	40

Consistency/Completeness: Same as above.

V[change_ok](T) =

conditions	values
$\text{parse}(T, I, C, CE) \wedge C = _ \wedge CE = _$	%undefined%
$\text{parse}(T, I, C, CE) \wedge C \neq _ \wedge CE = _$	1
else	0

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- cases one and two are distinguished by C test; else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- all outputs are constant or error and therefore defined.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- true.

$V[(output_screen)](T) =$

conditions	values
$T = Map()$	%no_output%
$parse(T, I, C, CE, N) \wedge$ $I = Map().init_draw(d,xw) \wedge$ $C = _$	rect of size 12*40 x 12*40 drawn in window with Display *d, Window xw
$parse(T, I, C, CE, N) \wedge$ $I = Map().init_draw(d,xw) \wedge$ $C = change_size(f,w,l,s)$	rect of size w*s x l*s drawn in window with Display *d, Window xw

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- If trace does not have just Map(), then I will be equal to Map().init_draw combination, and C comparison insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- values are either constant or defined from variables present in LHC, therefore defined.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- case 3 may be same as constant values in case two, but not always, requiring partitioning.

RETURN VALUES

Program Name	Argument No	Value
change_size	Value	change_ok

Completeness (2): There is one output function/relation that specifies each output value:

- There is one output value V[change_size] defined above for the one value in the table.

CLASS: POPUPMAPSIZE

TYPE IMPLEMENTED: <PopupMapSize>

(1) SYNTAX

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
init	<void>	<Xv_opaque> owner_frame	<Map*> pMap
show	<void>		
change	<void>	<Panel_item> item	

OUTPUT VARIABLES

Variable Name	Type	Access
(popup window)	(XView Popup window)	N/A

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
cfChange	<void>	<Panel_item>	<Event>

INPUT VARIABLES

Variable Name	Type	Access
change_error	<int>	input pseudo-event
Map::width	<double>	direct access
Map::length	<double>	direct access
Map::scale	<int>	direct access
entered_width	<char *>	XView xv_get value
entered_length	<char *>	XView xv_get value
entered_scale	<int>	XView xv_get value
FRAME_CMD_PUSHPIN_IN	<int>	XView xv_get value
XV_KEY_DATA	<Xv_opaque>	XView xv_get value
INSTANCE	<Attr_attribute>	direct access

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4
Map:: change_size	<int> change_error	<Xv_opaque> popup_frame	<double> new_width	<double> new_length	<int> new_scale

(2) CANONICAL TRACES

$$\begin{aligned}
 \text{canonical}(T_i) \leftrightarrow & (T_i = _) \vee (T_i = \text{init}(o,p)) \vee (T_i = \text{init}(o,p).\text{show}()) \vee (T_i = \text{init}(o,p).\text{show}().\text{change}(it)) \\
 & \vee \\
 & (T_i = \text{init}(o,p).\text{show}().\text{change}(it).\text{change_error}) \\
 \text{canonical}(T_c) \leftrightarrow & (T_c = _)
 \end{aligned}$$

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

AUXILIARY FUNCTIONS

parse(S,S1,S2,S3,S4) =

conditions	equivalences
$(S = S1.S2.S3.S4) \wedge$ $(S1 = [\text{init}(o,p)]_{i=0}^1) \wedge$ $(S2 = [\text{show}()]_{i=0}^1) \wedge$ $(S3 = [\text{change}(it)]_{i=0}^1) \wedge$ $(S4 = [\text{change_error}]_{i=0}^1)$	true
else	false

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for init, show, change, change_error, and cfChange

T.init(o,p) =

conditions	equivalences
T = _	init(o,p)
T ≠ _	%already_initialized%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- If one LHC condition is true, the other must be false, and they therefore partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- init(o,p) is defined by event itself, other RHC item is error message.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- Only one is specified, init(), and it is canonical.

Consistency (3): All RHC values are unique:

- One is value, one is error.

T.show() =

conditions	equivalences
T = _	%uninitialized%
else	I.show() where parse(T, I, S, C, CE)

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- First case is empty trace, second has something in trace, third is else, insuring partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- First two cases are errors, in last I must be defined since T is not empty.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- Trace init().show() [I.show()] is in the canonical trace.

Consistency (3): All RHC values are unique:

- True.

T.change(it) $\equiv$

conditions	equivalences
T = _	%uninitialized%
T = init(o,p)	%undisplayed%
parse(T, I, S, C, CE) $\wedge$ S $\neq$ _ $\wedge$ I=init(o,p) $\wedge$ p->change_size() = TRUE	equivalence = I.S.change(it); ADD-TO-TRACE(T _p , change_size(f, atof(entered_width), atof(entered_length), entered_scale)) where f is frame created by init()
else	equivalence = I.S.change(it).change_error; ADD-TO-TRACE(T _p , change_size(f, atof(entered_width), atof(entered_length), entered_scale)) where parse(T, I, S, C, CE) $\wedge$ I=init(o,p) $\wedge$ change_error = change_size() $\wedge$ f is frame created by init()

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- First two are obviously different, third has show() in T, else separates third from fourth.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- I and S defined for traces in 3rd/4th cases; p, change_error defined as given; entered* values defined if popup has been created (since init must be in trace, that is true).

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- init().show().change() and init().show().change().change_error are canonical

Consistency (3): All RHC values are unique:

- 3rd + 4th cases differ in equivalence

T.change_error $\equiv$

conditions	equivalences
T = init(o,p).show().change(it)	T.change_error
else	%undefined%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T defined by LHC.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T.change_error canonical if T is as defined by LHC

Consistency (3): All RHC values are unique:

- One value, one error.

T_C.cfChange(item,e) $\equiv$ T_C: ADD-TO-TRACE(T_p, change(item))
where PopMapSize* p = xv_get(item, XV_KEY_DATA, INSTANCE);

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- If change occurs, the PopMapSize object must have already been created, and p will be valid.

Consistency (2): All traces specified in the RIHC of the equivalence section are canonical:

- T_C canonical by definition.

- Consistency (3): All RHC values are unique:
- Only one value.

(4) VALUES

OUTPUT VALUES

$V[\text{popup_frame}](T) =$

conditions	values
$T = _$	%undefined%
else	frame created via init function

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- frame is defined by init, which must be part of any non-empty trace.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- Either frame or error.

$V[(\text{popup_window})](T) =$

conditions	values
$T = _$	%undefined%
$T = \text{init}(o,p)$	%undisplayed%
$T = \text{init}(o,p).\text{show}()$	popup window displayed on screen; Width field = $p \rightarrow \text{width}$ formatted "%.2f"; Length field = $p \rightarrow \text{length}$ formatted "%.2f"; Scale field = $p \rightarrow \text{scale}$; values may be modified by user
$T = \text{init}(o,p).\text{show}().\text{change}(it) \wedge$ $xv_get(\text{frame created by init}(),$ $\text{FRAME_CMD_PUSHPIN_IN}) = \text{TRUE}$	popup fields set to values from $p \rightarrow$ as given above
$T = T1.\text{change}(it) \wedge$ $xv_get(\text{frame created by init}(),$ $\text{FRAME_CMD_PUSHPIN_IN}) = \text{FALSE}$	popup window disappears from screen
else	popup forced to remain on screen, with values as modified by user

—○
Map Size

Width: <f%.2f>
Length: <f%.2f>

Scale: __<int>__

☒ ☒

Change

[PopupMapSize]

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- LHC partitions the entire canonical trace.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- window and fields are created by init, which is included in RHC trace

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- Either has constant (default) appearance or one modified by user input.

RETURN VALUES

(none)

CLASS: WINMAP

TYPE IMPLEMENTED: <WinMap>

(1) SYNTAX

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3
init	<void>	<Xv_opaque>	<Map*>	<PopupMapSize*>
unimplemented	<void>			
quit	<void>			

OUTPUT VARIABLES

Variable Name	Type	Access
frame	<Xv_opaque>	publicly accessible
(main window)	(XView window)	N/A

CLASS OUTPUT VARIABLES

Variable Name	Type	Access
notify_value	<Notify_value>	func return

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3
cfMenuFileQuit	<Menu_item>	<Menu_item>	<Menu_generate>	
cfMenuMapRedraw	<Menu_item>	<Menu_item>	<Menu_generate>	
cfMenuMapChangeSize	<Menu_item>	<Menu_item>	<Menu_generate>	
cfMenuUnimplemented	<Menu_item>	<Menu_item>	<Menu_generate>	
cfRepaint		<Canvas>	<Xv_window>	<Display>
		#4 <Window>	#5 <Xv_xrectlist>	
cfDestroy	<Notify_value>	<Xv_opaque>	<Destroy_status>	

INPUT VARIABLES

Variable Name	Type	Access
XV_KEY_DATA	<Xv_opaque>	XView xv_get value
INSTANCE	<Attr_attribute>	direct access

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3
PopupMapSize::show	<void>			
Map::init_draw	<void>	<Display> display	<Window> xid	
Utils::cfNotice_OK	<void>	<Xv_opaque> owner	<char *> message	

(2) CANONICAL TRACES

$$\text{canonical}(T_i) \leftrightarrow (T_i = _) \vee (T_i = \text{init}(o, p1, p2))$$

$$\text{canonical}(T_c) \leftrightarrow (T_c = _) \vee (T_c = \text{cfDestroy}(c, s))$$

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for access functions init, unimplemented, quit, cfMenuFileQuit, cfMenuMapRedraw, cfMenuMapChangeSize, cfRepaint, and cfDestroy

T.init(o,p1,p2) =

conditions	equivalences
T = _	equivalence= init(o,p1,p2); ADD-TO-TRACE(T _{p1} , init_draw(disg, xid)) where disg + xid are defined by XView calls to create the window
else	%already_initialized%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- init(o,p) is defined by event itself, other RHC item is error message.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- Only one is specified, init(), and it is canonical.

Consistency (3): All RHC values are unique:

- One value, one error.

T.unimplemented() =

conditions	equivalences
T = _	%uninitialized%
else	equivalence = T; ADD-TO-TRACE(T _u , cfNotice_OK(f, "This function has not been implemented.") where T _u is the class access trace for Utils

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T defined by equivalence.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T is canonical by definition.

Consistency (3): All RHC values are unique:

- One value, one error.

T.quit() =

conditions	equivalences
T = _	%uninitialized%
else	T

Completeness/Consistency same as above.

$T_C.cfMenuFileQuit(item, op) \equiv$

conditions	equivalences
$xv_get(item, XV_KEY_DATA, INSTANCE) = 0$	%invalid item%
$op = MENU_NOTIFY \wedge$ $xv_get(item, XV_KEY_DATA, INSTANCE) \neq 0$	equivalence = T_C ; ADD-TO-TRACE(T_p , quit()) where $WinMap^* p =$ $xv_get(item, XV_KEY_DATA, INSTANCE)$;
else	T_C

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first two differentiated by $\neq$; else insure partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T_C defined by LHS; if fn called then item must be created and therefore p will be valid.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T_C is canonical by definition.

Consistency (3): All RHC values are unique:

- error + one has ADD-TO-TRACE, other does not.

$T_C.cfMenuMapRedraw(item, op) \equiv$

conditions	equivalences
$xv_get(item, XV_KEY_DATA, INSTANCE) = 0$	%invalid item%
$op = MENU_NOTIFY \wedge$ $xv_get(item, XV_KEY_DATA, INSTANCE) \neq 0$	equivalence = T_C ; ADD-TO-TRACE(T_p , draw()) where $Map^* p =$ $xv_get(item, XV_KEY_DATA, INSTANCE)$;
else	T_C

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first two differentiated by $\neq$; else insure partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T_C defined by LHS; if fn called then item must be created and therefore p will be valid.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T_C is canonical by definition.

Consistency (3): All RHC values are unique:

- error + one has ADD-TO-TRACE, other does not.

$T_C.cfMenuMapChangeSize(item, op) \equiv$

conditions	equivalences
$xv_get(item, XV_KEY_DATA, INSTANCE) = 0$	%invalid item%
$op = MENU_NOTIFY \wedge$ $xv_get(item, XV_KEY_DATA, INSTANCE) \neq 0$	equivalence = T_C ; ADD-TO-TRACE(T_p , show()) where $PopupMapSize^* p =$ $xv_get(item, XV_KEY_DATA, INSTANCE)$;
else	T_C

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first two differentiated by $\neq$; else insure partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T_C defined by LHS; if fn called then item must be created and therefore p will be valid.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T_C is canonical by definition.

Consistency (3): All RHC values are unique:

- error + one has ADD-TO-TRACE, other does not.

$T_C.cfMenuUnimplemented(item, op) \equiv$

conditions	equivalences
$xv_get(item, XV_KEY_DATA, INSTANCE) = 0$	%invalid item%
$op = MENU_NOTIFY \wedge$ $xv_get(item, XV_KEY_DATA, INSTANCE) \neq 0$	equivalence = T_C ; ADD-TO-TRACE(T_p , unimplemented()) where $WinMap^* p =$ $xv_get(item, XV_KEY_DATA, INSTANCE)$;
else	T_C

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first two differentiated by $\neq$; else insure partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T_C defined by LHS; if fn called then item must be created and therefore p will be valid.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T_C is canonical by definition.

Consistency (3): All RHC values are unique:

- error + one has ADD-TO-TRACE, other does not.

$T_C.cfRepaint(canvas, pw, display, xid, rects) \equiv$

conditions	equivalences
$xv_get(pw, XV_KEY_DATA, INSTANCE) = 0$	%invalid item%
$op = MENU_NOTIFY \wedge$ $xv_get(pw, XV_KEY_DATA, INSTANCE) \neq 0$	equivalence = T_C ; ADD-TO-TRACE(T_p , draw()) where $Map^* p =$ $xv_get(pw, XV_KEY_DATA, INSTANCE)$;
else	T_C

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first two differentiated by $\neq$; else insure partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T_C defined by LHS; if fn called then item must be created and therefore p will be valid.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T_C is canonical by definition.

Consistency (3): All RHC values are unique:

- error + one has ADD-TO-TRACE, other does not.

$T_C.cfDestroy(client, status) \equiv cfDestroy(client, status)$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- no partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by LHS

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- $cfDestroy()$ is canonical

Consistency (3): All RHC values are unique:

- one value only

(4) VALUES

OUTPUT VALUES

V[frame](T) =

conditions	values
T = _	%undefined%
T = init(o,p1,p2)	frame id for (main window)

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- LHC partitions the canonical trace.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- frame is defined by init.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

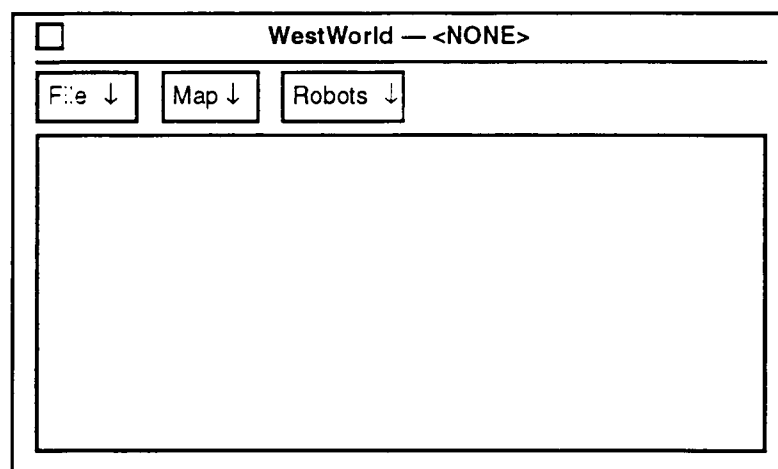
- No traces in RHC.

Consistency (3): All RHC values are unique:

- Either frame or error.

V[(main_window)](T) =

conditions	values
T = _	%undefined%
T = init(o,p1,p2)	display WinMap on screen, with canvas exactly encompassing default map size, with title "WestWorld -- <None>", with border fitting map size (Map::init_draw), with menus as follows: - File: Load..., Save..., Quit - Map: Redraw <default>, <blank>, Update HELIX Map, Clear Map, Change Map Size..., New Map Object... - Robots: Summon... <default>, <blank>, Start All, Stop All, Quit All



Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- LHC partitions the canonical trace.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- window is defined by init.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC.

Consistency (3): All RHC values are unique:

- Either window or error.

$V[\text{notify_value}](T_C) =$

conditions	values
$T_C = _$	%undefined%
$T_C = \text{cfDestroy}(\text{client}, \text{status}) \wedge$ $\text{status} = \text{DESTROY_CLEANUP}$	$\text{notify_next_destroy_func}(\text{client}, \text{status})$
else	NOTIFY_DONE

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first two differ, else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- constant, error, or client/status defined by LHC

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- Error, constant, or fn call return

RETURN VALUES

Program Name	Argument No	Value
cfDestroy	Value	notify_value

Completeness (2): There is one output function/relation that specifies each output value:

- There is one output value defined above for notify_value.

CLASS: UTILS

TYPE IMPLEMENTED: <Utils>

(1) SYNTAX

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4
cfNotice_OK	(notice) + <void>	<Xv_opaque> owner	<char *> message		

CLASS OUTPUT VARIABLES

Variable Name	Type	Access
(notice)	(XView Notice)	N/A

INPUT VARIABLES

Variable Name	Type	Access
notice_confirm	notice Confirm button	pseudo-event

(2) CANONICAL TRACES

$$\text{canonical}(T_C) \leftrightarrow (T_C = _) \vee (T_C = \text{cfNotice_OK}(o,m))$$

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for cfNotice_OK and notice_confirm.

$T_C.\text{cfNotice_OK}(o,m) \equiv$

conditions	equivalences
$T_C = \text{cfNotice_OK}(o,m)$	%waiting%
else	cfNotice_OK(o,m)

Completeness (3): The predicates in the LHIC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHIC are defined whenever the corresponding predicate in the LHC is 'true':

- event defined by LHC.

Consistency (2): All traces specified in the RHIC of the equivalence section are canonical:

- cfNotice_OK is canonical.

Consistency (3): All RHC values are unique:

- One value, one error.

$T_C.\text{notice_confirm} \equiv$

conditions	equivalences
$T_C = \text{cfNotice_OK}(o,m)$	_
else	%no notice%

Completeness (3): The predicates in the LHIC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- constants.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- `_` is canonical.

Consistency (3): All RHC values are unique:

- One value, one error.

(4) VALUES

OUTPUT VALUES

$V[(notice)](T) =$

conditions	values
$T = _$	<code>%no_output%</code>
$T = cfNotice_OK(o, m)$	display XView notice with owner o, message m, and Confirm button

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Since T is canonical, the conditions partition the canonical trace and therefore give a full partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by LHC

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- One value, one error.

RETURN VALUES

<none>

IV. Clear Boxes

The clear boxes consist of the following files, which are attached:

- ww_ui.H — header file containing class, constant, and misc. definitions
- Main.C — file with main() loop and global variables
- Map.C — class implementation for Map
- WinMap.C — class implementation for WinMap
- PopupMapSize.C — class implementation for PopupMapSize
- Utils.C — misc/utility routines

Increment 1 C++ Header Definitions

```
// ww_ui.H
//
// WestWorld
//
// Alex L. Bangs, 2/10/93
//-----
// Modification History:
// 2/10/93 ALB Increment 1

#ifndef WW_UI_HEADER
#define WW_UI_HEADER

#include <math.h>

// Map constants

const double default_width = 12.0;
const double default_length = 12.0;
const int default_scale = 40;
const int min_scale = 1;
const int max_scale = 100;
const double min_width = 1.0;
const double min_length = 1.0;
const int panel_text_size = 80;

// simple #define functions

#define min(a,b) ((a) < (b) ? (a) : (b))
#define scaleIt(coord) (rint((coord) * scale))

// Main descriptor
// (note no real class for Main, but has function + globals
// class Main
// void main(int argc, char **argv);
extern Attr_attribute INSTANCE;

// Other class descriptors
class Map {
    Display *display;
    Window xid;
    GC gc;

public:
    double width, length;
    int scale;

    Map();
    void init_draw(Display*, Window);
```

```

void draw();
int change_size(Xv_opaque frame,
               double new_width, double new_length, int new_scale);
};

```

```

class PopupMapSize {
    Xv_opaque    frame;
    Xv_opaque    controls;
    Xv_opaque    map_width_field;
    Xv_opaque    map_length_field;
    Xv_opaque    map_scale_field;
    Xv_opaque    change_button;

    Map*    pMap;
    void    update();    // update numbers in the window

```

```

    public:
        void    init(Xv_opaque owner, Map* pTheMap);
        void    show();    // redisplay the box, and do an update
        void    change(Panel_item); // change button pressed; send values to pMap

    // class functions
        static void cfChange(Panel_item item, Event *event);
        // XView button callback for Change
};

```

```

class WinMap {
    Xv_opaque    controls;
    Xv_opaque    file_menu_button;
    Xv_opaque    map_menu_button;
    Xv_opaque    robots_menu_button;
    Xv_opaque    canvas;
    Xv_window    canvas_paint;
    Display*    display;
    Window xid;

    Xv_opaque    file_menu_create(caddr_t *, Xv_opaque);
    Xv_opaque    map_menu_create(caddr_t *, Xv_opaque);
    Xv_opaque    robots_menu_create(caddr_t *, Xv_opaque);

    Map*    pMap;
    PopupMapSize* pPopupMapSize;

```

```

    public:
        Xv_opaque    frame;

        void    init(Xv_opaque owner, Map*, PopupMapSize*);
        void    unimplemented();
        void    quit();

        static Menu_item cfMenuFileQuit(Menu_item item, Menu_generate op);
        static Menu_item cfMenuMapRedraw(Menu_item item, Menu_generate op);
        static Menu_item cfMenuMapChangeSize(Menu_item item, Menu_generate op);
        static Menu_item cfMenuUnimplemented(Menu_item item, Menu_generate op);

    // general XView callbacks
        static Notify_value cfDestroy(Xv_opaque client, Destroy_status status);
        static void    cfRepaint(Canvas canvas, Xv_window paint_window,
                                Display *display, Window xid,
                                Xv_xrectlist *rects);

```

```
};  
  
class Utils {  
public:  
    static void cfNotice_OK(Xv_opaque owner, char* message);  
};  
  
#endif
```

Appendix D: The Second Increment Specification

WestWorld

Increment 2 Specification

I. Top Level Black Box

define_BB void WestWorld
input

Invocation
XCB:Menu/File/Load
XCB:Menu/File/Save
XCB:Menu/File/Quit
XCB:Menu/Map/Redraw
XCB:Menu/Map/UpdateHelix
XCB:Menu/Map/Clear
XCB:Menu/Map/ChangeSize
XCB:Menu/Map/NewObj
XCB:Menu/Robots/Summon
XCB:Menu/Robots/Start
XCB:Menu/Robots/Stop
XCB:Menu/Robots/Quit
XCB:Button/MapLoad
XCB:Button/MapSave
XCB:Button/MapSizeChange
XCB:XV/Destroy
XCB:XV/Repaint

output

window WinMap
popup PopupLoadMap
popup PopupSaveMap
popup PopupMapSize
notice Notice_OK
notice Notice_YN

transition

Invocation -->
 _WinMapInit()
XCB:Menu/File/Load -->
 _PopupLoadMapShow()
XCB:Menu/File/Save -->
 _PopupSaveMapShow()
XCB:Menu/File/Quit -->
 xv_destroy_safe(WinMap frame created at Invocation)
XCB:Menu/Map/Redraw -->
 _Repaint()
XCB:Menu/Map/UpdateHelix -->
 _Unimplemented(WinMap frame created at Invocation);
XCB:Menu/Map/Clear -->
 _Repaint();
XCB:Menu/Map/ChangeSize -->
 _PopupMapSizeShow()
XCB:Menu/Map/NewObj -->
 _Unimplemented(WinMap frame created at Invocation);
XCB:Menu/Robots/Summon -->

```

        _Unimplemented(WinMap frame created at Invocation);
XCB:Menu/Robots/Start -->
        _Unimplemented(WinMap frame created at Invocation);
XCB:Menu/Robots/Stop -->
        _Unimplemented(WinMap frame created at Invocation);
XCB:Menu/Robots/Quit -->
        _Unimplemented(WinMap frame created at Invocation);
XCB:Button/MapSizeChange -->
        _PopupMapSizeChange()
XCB:Button/MapLoad -->
        _PopupLoadMapLoad()
XCB:Button/MapSave -->
        _PopupSaveMapSave()
XCB:XV/Destroy(client, status) -->
        _Destroy(client, status)
XCB:XV/Repaint -->
        _Repaint()
end_BB

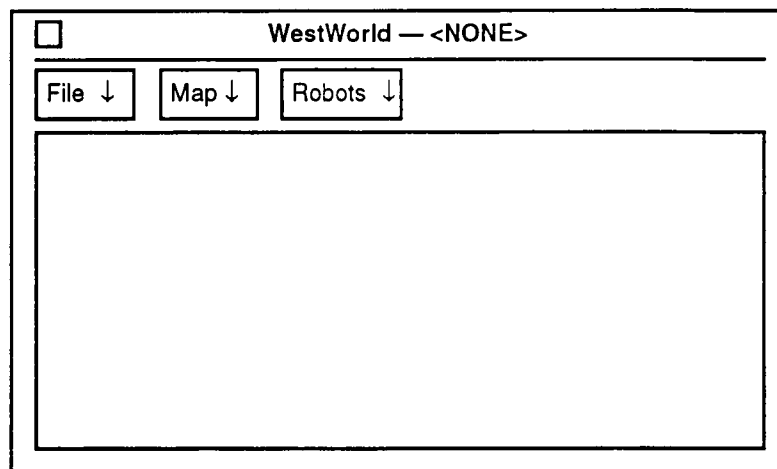
```

Black Box Specification Functions:

```

[ _WinMapInit() ] ≡
[
    display WinMap on screen, with canvas exactly encompassing default map size
    with title "WestWorld -- <None>"
    with menus as follows:
        File: Load..., Save..., Quit
        Map: Redraw <default>, <blank>, Update HELIX Map,
            Clear Map, Change Map Size..., New Map Object...
        Robots: Summon... <default>, <blank>, Start All,
            Stop All, Quit All
    with border fitting map size (_Repaint)

```



```

    ]
[ _Repaint() ] ≡
[
    clear WinMap paint window created at Invocation and draw map border using X
    Display+Window params for paint window given width, length, scale from default or
    XCB:PopupMapSizeChange; draw map objects since last successful load, if not clear since
    last load
]

```

```
[ _Destroy(Xv_opaque client, Destroy_status status) ] =
[ status = DESTROY_CHECKING --> NOTIFY_DONE
| status = DESTROY_SAVE_YOURSELF --> NOTIFY_DONE
| status = DESTROY_CLEANUP --> notify_next_destroy_func(client, status)
| status = DESTROY_PROCESS_DEATH --> NOTIFY_DONE
]
```

```
[ _PopupLoadMapShow() ] =
[ display PopupLoadMap with either last file loaded or saved (whichever was most recent) or
blank
```

```
]
```

```
[ _PopupLoadMapLoad() ] =
[ if open file on filename from PopupLoadMap results in error
--> _Notice_OK(PopupLoadMap frame, "The file could not be opened.")
| for each line in file
if in correct format, add object to map
else _Notice_OK(PopupLoadMap frame, "Map file format error: <error> @ line
<line>")
where error is the form of the error (possible errors are "bad box definition",
"bad cylinder definition:, and "unknown object") and <line> is the line where it
occured
change title on window created by _WinMapInit() to include last component of filename
_Repaint
]
```

```
[ _PopupSaveMapShow() ] =
[ display PopupSaveMap with either last file loaded or saved (whichever was most recent) or
blank
```

```
]
```

```
[ _PopupSaveMapSave() ] =
[ ((file already exists ^ _Notice_YN(PopupSaveMap frame, "File exists. Overwrite it?")) v
file doesn't exist) ^ no file write/open errors -->
write comment line to file with file name and date/time of write;
for each object in map from load process since clear, write line to file
| file open error -->
_Notify_OK(PopupSaveMap frame, "File could not be opened for write.")
| file write error -->
_Notify_OK(PopupSaveMap frame, "An error occurred writing the file.")
]
```

```
[ _PopupMapSizeShow() ] =
[ display PopupMapSize with map width (using %.2f format) of default of 12 m or last width
  set by successful XCB:Button/MapSizeChange and length (using %.2f format) of 12m or last
  height set by XCB:Button/MapSizeChange and scale of default 40 or last scale set by
  XCB:Button/MapSizeChange
```

```
]
```

```
[ _PopupMapSizeChange() ] =
[ (atof(entered width) < 1 | atof(entered length) < 1) -->
  Notice_OK(PopupMapSize frame,
    "Width and Length must be at least 1.0m.")
| (entered scale < 1) | (entered scale > 100)
  Notice_OK(PopupMapSize frame,
    "Scale must be in the range of 1 to 100.")
| (PopupMapSize pushpin is in) -->
  redisplay values in PopupMapSize;
  _Repaint()
| true -->
  _Repaint()
]
```

```
[ _Unimplemented(Xv_opaque owner) ] =
[ _Notice_OK(owner, "This function has not been implemented.") ]
```

```
[ _Notice_OK(Xv_opaque owner, char *message) ] =
[ display notice for owner
  with Confirm button and given message string:
```

```
]
```

```
[ _Notice_YN(Xv_opaque owner, char *message) ] =
[ display notice for owner
  with Yes/No buttons and given message string;
  return TRUE (1) if Yes pressed, FALSE (0) otherwise
```

```
]
```


II. Class Design and Class BB Specifications

(1) Choose candidate objects

The WinMap, PopupMapSize, Map, Utils, and Main classes were already defined in increment 1. Increment 2 adds two new popup windows, so PopupLoadMap and PopupSaveMap classes need to be defined. Consideration of an abstract base class called Popup to hold the common interface and data for all the popup objects may be in order. Handling files could be pushed into a new class, but it is best at this point to use basic functions from UNIX (fopen, etc) and have Map be responsible for the filename.

(2) Assign top-level stimuli to objects

Main

Invocation

Map

<none>

PopupLoadMap

XCB:Button/MapLoad

PopupSaveMap

XCB:Button/MapSave

PopupMapSize

XCB:Button/MapSizeChange

WinMap

XCB:Menu/File/Load

XCB:Menu/File/Save

XCB:Menu/File/Quit

XCB:Menu/Map/Redraw

XCB:Menu/Map/UpdateHelix

XCB:Menu/Map/Clear

XCB:Menu/Map/ChangeSize

XCB:Menu/Map/NewObj

XCB:Menu/Robots/Summon

XCB:Menu/Robots/Start

XCB:Menu/Robots/Stop

XCB:Menu/Robots/Quit

XCB:XV/Destroy

XCB:XV/Repaint

Utils

<none>

(3) Identify inter-class stimuli

Main, via the main() function, will have to initialize/create all the other objects, through either init calls or constructors (same as increment 1).

Map will draw the map info in the paint window of WinMap, based on window information passed from WinMap. It will assume a default map size until a map size change is sent from another object. It should have the basic map parameters publicly available. For increment 2, the filename must be available, and functions called load(Xv_opaque frame, char *filename) and save(Xv_opaque frame, char *filename)

which return TRUE if successful and 0 if not. If successful, they set the title bar of the WinMap window to reflect the current filename loaded. clear() clears out the map. load() will read in lines from the data file and call itself via loadline() for each line in the Map. The loadline() function is responsible for interpreting the file data as objects or comments or errors. Eventually, loadline will have to have some data structure for storing this data for the Map, but this must be discovered at the clear-box level.

PopupLoadMap will draw the Load Map popup and handle the button callback for that popup. It will display the current value of the filename held by the Map when show() is called. It will call Map::load() when the Load button is pressed. PopupSaveMap is basically the same.

PopupMapSize will draw the Change Map Size popup and handle the button callback for that popup. It will have to be able to display the popup on command when the appropriate menu item is selected via WinMap, and it will have to pass the change size parameters to Map when the Change button is pressed (same as increment 1).

Since the popup objects all share some common traits, including functions for show() and init(), an abstract superclass called Popup might be considered to define the common interface and functions that must be defined by the subclasses. However, the init() functions may be different for each, and the composition of popup windows may be significantly different, so a superclass will not be considered at this time.

WinMap will be responsible for drawing the main window. It will accept all menu callbacks, but will only handle those directly related to what it controls. Since Map will handle the drawing of the map in a subpane of WinMap, data referencing that subpane will have to be passed to Map, as will the actual draw calls. The menu item that causes the Change Map Size popup to appear will have to be passed to PopupMapSize. For incr 2, we have to pass show calls to PopupLoadMap and PopupSaveMap, and pass the clear function to Map(). In addition, when Map::init_draw() is called, a reference to the WinMap window is required which will allow Map to change the title of the window to reflect the latest loaded/saved filename.

Utils will handle the cfNotice_OK call. For increment 2, we add cfNotice_YN. It does not have any instantiations.

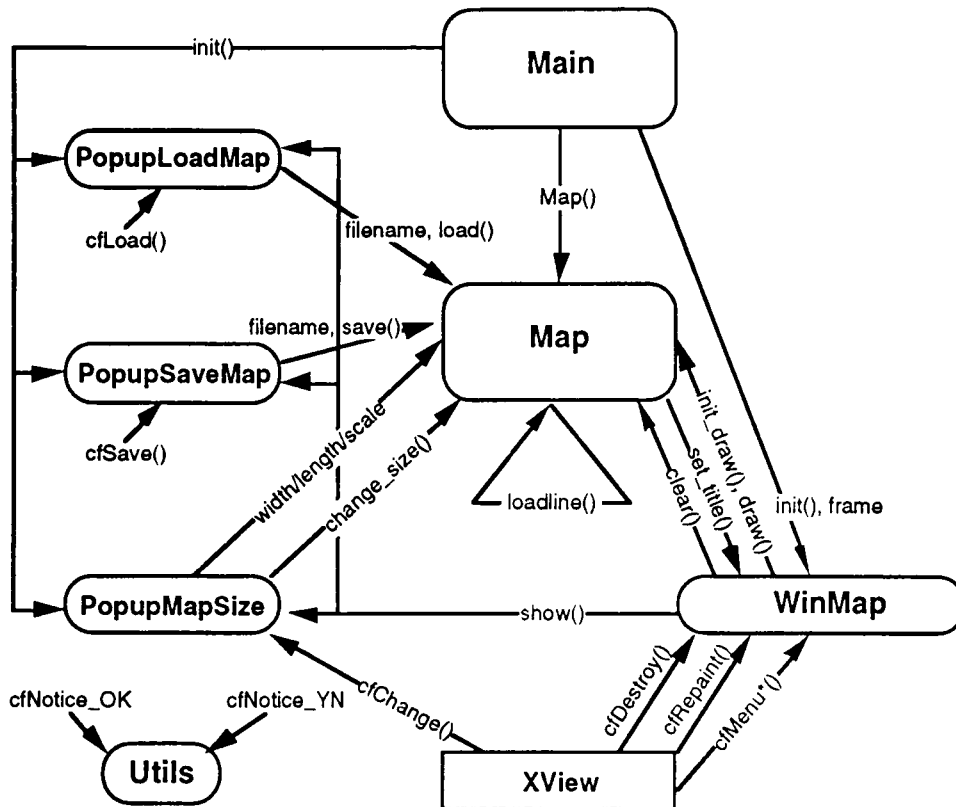


Figure X: Second Increment Object Interaction Diagram

(4) Black Box Definitions

define_BB Main

class access programs
 <Invocation>
 main()
 <Exit>

class output variables
 Attr_attribute INSTANCE

input variables
 Xv_opaque WinMap::frame

external access

Map::Map()
 Map::~~Map()
 void PopupLoadMap::init(Xv_opaque owner_frame, Map* pMap)
 void PopupSaveMap::init(Xv_opaque owner_frame, Map* pMap)
 void PopupMapSize::init(Xv_opaque owner_frame, Map* pMap)
 void WinMap::init(Xv_opaque owner_frame, Map* pMap,
 PopupLoadMap* pPopupLoadMap, PopupSaveMap* pPopupSaveMap,
 PopupMapSize* pPopupMapSize)

transition

$S_i = \langle \text{Invocation} \rangle \rightarrow$

```

        create Map [invokes Map()], PopupLoadMap, PopupSaveMap, PopupMapSize,
        WinMap
    Si = main() -->
        call init for PopupLoadMap, PopupSaveMap, PopupMapSize + WinMap object
        created by Invocation
    Si = <Exit> -->
        destroy Map [invokes ~Map()]

end_BB

define_BB Map
    access programs
        Map()
        ~Map()
        void init_draw(Display *display, Window xid, WinMap* pWinMap)
        void draw()
        void clear()
        int change_size(Xv_opaque frame, double new_width,
            double new_length, int new_scale)
        int load(Xv_opaque frame, char *loadfile)
        int save(Xv_opaque frame, char *savefile)
        int loadline(Xv_opaque frame, int lineno, char *line)

    output variables
        double width
        double length
        double scale
        char *filename

    output
        X display window
        XView notice

    external access
        void Utils::cfNotice_OK(Xv_opaque frame, char* message)
        int Utils::cfNotice_YN(Xv_opaque frame, char* message)
        void WinMap::set_title(char *)
        fopen, fclose, fgets, fputs [stdio calls]

    transition
        Si = Map() --> No response.
        Si = ~Map() --> No response.
        Si = init_draw(display, xid, pWinMap) -->
            clear window and draw map border (rectangle) of 12*40 x 12*40 [call draw()],
            using given display + xid parameters.
        Si = draw() -->
            clear window and draw map border (rectangle) of w*s x l*s where w,l,s are
            default or from last legal change_size(), using display + xid parameters from
            previous init_draw() call; for each object in a legal loadline() call since last
            Map() or clear() or load(), draw appropriate object with from init_draw
            display+xid parameters;
        Si = clear() --> call draw() for self, set filename in title to <NONE>
        (Si = change_size(frame, w, l, s)) ^ ((w < 1) ∨ (l < 1)) -->
            Utils::cfNotice_OK(frame, "Width and Length must be at least 1.0m.");
            return FALSE value from change_size
        (Si = change_size(frame, w, l, s)) ^ ((s < 1) ∨ (s > 100)) -->
            Utils::cfNotice_OK(frame, "Scale must be in the range of 1 to 100.")

```

```

        return FALSE value from change_size
(Si = change_size(frame, w, l, s)) ∧ change_valid(w, l, s) -->
    call draw() for self;
    return TRUE value from change_size
(Si = load(frame, f)) ∧ error opening f -->
    Utils::cfNotice_OK(frame, "The file could not be opened.")
    return FALSE from load()
(Si = load(frame, f)) ∧ error reading f -->
    Utils::cfNotice_OK(frame, "An error occurred reading the file.")
    return FALSE from load()
(Si = load(frame, f)) ∧ error closing f -->
    Utils::cfNotice_OK(frame, "An error occurred closing the file.")
    return FALSE from load()
(Si = load(frame, f)) ∧
(this->loadline(frame, line #, line from f) = FALSE) -->
    return FALSE from load()
(Si = load(frame, f)) ∧
(this->loadline(frame, line #, line from f) = TRUE) -->
    call pWinMap->set_title(ff) where ff is the file component from the path f
    for each line in file, call this->loadline(frame, line #, line)
    return TRUE from load()
(Si = save(frame, f)) ∧ f exists ∧
Utils::cfNotice_YN(frame, "File exists. Overwrite it?") = TRUE ∧ file create error -->
    Utils::cfNotice_OK(frame, "File could not be opened for write.")
    return FALSE from save
(Si = save(frame, f)) ∧ f exists ∧
Utils::cfNotice_YN(frame, "File exists. Overwrite it?") = FALSE -->
    return FALSE from save
(Si = save(frame, f)) ∧ error opening f -->
    Utils::cfNotice_OK(frame, "File could not be opened for write.")
    return FALSE from save
(Si = save(frame, f)) ∧ no errors opening f ∧ error writing f -->
    Utils::cfNotice_OK(frame, "An error occurred writing the file.")
    return FALSE from save
(Si = save(frame, f)) ∧ no errors opening f ∧ no error writing f -->
    write file with heading of file name + date + time of save,
    one line for each legal loadline() in stim. hist since last Map(), clear() or load();
    call pWinMap->set_title(ff) where ff is the file component from the path f
    return TRUE from save
Si = loadline(f,n,l) ∧ ((l[0] = '#') ∨ (l[0] = '\0')) --> TRUE
Si = loadline(f,n,l) ∧ legal_box(l) --> TRUE
Si = loadline(f,n,l) ∧ (strcmp(l, "box", 3) = 0) ∧ not(legal_box(l)) -->
    Utils::cfNotice_OK(f, "Map file format error: bad box definition @ line <n>")
    return FALSE
Si = loadline(f,n,l) ∧ legal_cylinder(l) --> TRUE
Si = loadline(f,n,l) ∧ (strcmp(l, "cyl", 3) = 0) ∧ not(legal_cylinder(l)) -->
    Utils::cfNotice_OK(f,
        "Map file format error: bad cylinder definition @ line <n>")
    return FALSE
Si = loadline(f,n,l) ∧ (strcmp(l, "box", 3) ≠ 0) ∧ (strcmp(l, "cyl", 3) ≠ 0) ∧ (l[0] ≠ '#')
--> Utils::cfNotice_OK(f, "Map file format error: unknown object @ line
<n>");
    return FALSE
(Si = filename) ∧ (no successful load) ∧ (no successful save) --> ""
(Si = filename) ∧ (∃Sj | (j < i) ∧

```

```

(((Sj = load(fr,f)) ∧ (load ok)) ∨ ((Sj = save(fr,f) ∧ (save ok))) ∧
not(∃Sk | (j < k < i) ∧ (((Sk = load(fr,f)) ∧ (load ok)) ∨ ((Sk = save(fr,f) ∧ (save ok))) –
>
f
Si = width ∧ (∃Sj | (j < i) ∧ (Sj = change_size(f,w,l,s) ) ∧ change_valid(w,l,s)) ∧
not(∃Sk | (j < k < i) ∧ (Sk = change_size(f,w,l,s') ) ∧ change_valid(w',l',s')) --> w
Si = width ∧ not(∃Sj | (j < i) ∧ (Sj = change_size(f,w,l,s) ) ∧ change_valid(w,l,s)) --> 12
Si = length ∧ (∃Sj | (j < i) ∧ (Sj = change_size(f,w,l,s) ) ∧ change_valid(w,l,s)) ∧
not(∃Sk | (j < k < i) ∧ (Sk = change_size(f,w,l,s') ) ∧ change_valid(w',l',s')) --> 1
Si = length ∧ not(∃Sj | (j < i) ∧ (Sj = change_size(f,w,l,s) ) ∧ change_valid(w,l,s)) --> 12
Si = scale ∧ (∃Sj | (j < i) ∧ (Sj = change_size(f,w,l,s) ) ∧ change_valid(w,l,s)) ∧
not(∃Sk | (j < k < i) ∧ (Sk = change_size(f,w,l,s') ) ∧ change_valid(w',l',s')) --> s
Si = scale ∧ not(∃Sj | (j < i) ∧ (Sj = change_size(f,w,l,s) ) ∧ change_valid(w,l,s)) --> 40

```

end_BB

Spec Function

```

[ change_valid(w,l,s) ] ≡
[ ((1 ≤ s ≤ 100) ∧ (w ≥ 1) ∧ (l ≥ 1)) ]
[ legal_box(l) ] ≡
[ if l is of form "box <locx> <locy> <width> <length> [<height>]" --> TRUE
else --> FALSE ]
[ legal_cylinder(l) ] ≡
[ if l is of form "cyl[inder] <locx> <locy> <radius> [<height>]" --> TRUE
else --> FALSE ]

```

NOTES:

- (1) assumption is made that init_draw comes before any change_size; no error checking for this
- (2) Map() must be first stimuli, by default, since it is a constructor

define_BB PopupLoadMap

```

access programs
void init(Xv_opaque owner_frame, Map* pMap)
void show()
void load(Panel_item)

output
popup window

class access programs
static void clLoad(Panel_item, Event)

input variables
char *Map::filename;
Attr_attribute Main::INSTANCE;
xv_get variables FRAME_CMD_PUSHPIN_IN, XV_KEY_DATA

external access
int Map::load(Xv_opaque frame, char *loadfile)
char *Map::filename

transition
Si = init(o, p) --> no response.
Si = show() -->
display popup screen with owner o, with values filename field from
p->filename, where (∃Sj | (j < i) ∧ (Sj = init(o,p)))

```

$S_i = \text{load}(\text{item}) \rightarrow$
 given pointer to popup input field for filename and popup frame "f" created by
 $\text{init}(\exists S_j \mid (j < i) \wedge (S_j = \text{init}(o, p)))$, call $p \rightarrow \text{load}(f, \text{entered filename})$; if $p \rightarrow \text{load}()$ returns 1 and xv_get parameter `FRAME_CMD_PUSHPIN_IN` from f is 1, then call $\text{show}()$; if $p \rightarrow \text{load}()$ returns 0, send an error to XView via item to hold the popup on the screen.

$S_i = \text{cfLoad}(\text{item}, \text{ev}) \rightarrow$
 call `PopupLoadMap* p->load(item)` where $p = xv_get(\text{item}, \text{XV_KEY_DATA}, \text{INSTANCE})$

end_BB

NOTES:

(1) assumption is made that $\text{init}()$ comes before any other calls

define_BB PopupSaveMap

access programs

void $\text{init}(\text{Xv_opaque owner_frame}, \text{Map* pMap})$

void $\text{show}()$

void $\text{save}(\text{Panel_item})$

output

popup window

class access programs

static void $\text{cfSave}(\text{Panel_item}, \text{Event})$

input variables

char *Map::filename;

Attr_attribute Main::INSTANCE;

xv_get variables `FRAME_CMD_PUSHPIN_IN`, `XV_KEY_DATA`

external access

int Map::save(Xv_opaque frame, char *savefile)

char *Map::filename

transition

$S_i = \text{init}(o, p) \rightarrow$ no response.

$S_i = \text{show}() \rightarrow$

display popup screen with owner o, with values filename field from
 $p \rightarrow \text{filename}$, where $(\exists S_j \mid (j < i) \wedge (S_j = \text{init}(o, p)))$

$S_i = \text{save}(\text{item}) \rightarrow$

given pointer to popup input field for filename and popup frame "f" created by
 $\text{init}(\exists S_j \mid (j < i) \wedge (S_j = \text{init}(o, p)))$, call $p \rightarrow \text{save}(f, \text{entered filename})$; if $p \rightarrow \text{save}()$ returns 1 and xv_get parameter `FRAME_CMD_PUSHPIN_IN` from f is 1, then call $\text{show}()$; if $p \rightarrow \text{save}()$ returns 0, send an error to XView via item to hold the popup on the screen.

$S_i = \text{cfSave}(\text{item}, \text{ev}) \rightarrow$

call `PopupSaveMap* p->save(item)` where $p = xv_get(\text{item}, \text{XV_KEY_DATA}, \text{INSTANCE})$

end_BB

NOTES:

(1) assumption is made that init() comes before any other calls

define_BB PopupMapSize

access programs

```
void init(Xv_opaque owner_frame, Map* pMap)
void show()
void change(Panel_item)
```

output

popup window

class access programs

```
static void cfChange(Panel_item, Event)
```

input variables

```
Attr_attribute Main::INSTANCE;
xv_get variables FRAME_CMD_PUSHPIN_IN, XV_KEY_DATA, entered_width,
entered_length, entered_scale
```

external access

```
int Map::change_size(Xv_opaque frame, double new_width,
double new_length, int new_scale)
double Map::width
double Map::length
double Map::scale
```

transition

$S_i = \text{init}(o, p) \rightarrow$ no response.

$S_i = \text{show}() \rightarrow$

display popup screen with owner o, with values in width/length/scale fields from $p \rightarrow \text{width}$, $p \rightarrow \text{length}$, $p \rightarrow \text{scale}$, where $(\exists S_j \mid (j < i) \wedge (S_j = \text{init}(o, p)) \wedge \text{not}(\exists S_k \mid (j < k < i) \wedge (S_k = \text{init}(o, p))))$

$S_i = \text{change}(\text{item}) \rightarrow$

given pointer to popup input fields for width/length/scale and popup frame "f" created by init $(\exists S_j \mid (j < i) \wedge (S_j = \text{init}(o, p)))$, call $p \rightarrow \text{change_size}(f, \text{entered width}, \text{entered length}, \text{entered scale})$; if change_size returns 1 and xv_get parameter FRAME_CMD_PUSHPIN_IN from f is 1, then call show(); if change_size() returns 0, send an error to XView via item to hold the popup on the screen.

$S_i = \text{cfChange}(\text{item}, \text{ev}) \rightarrow$

call PopupMapSize* $p \rightarrow \text{change}(\text{item})$ where $p = \text{xv_get}(\text{item}, \text{XV_KEY_DATA}, \text{INSTANCE})$

end_BB**NOTES:**

(1) assumption is made that init() comes before any other calls

define_BB WinMap

access programs

```
void init(Xv_opaque owner_frame, Map* pMap,
PopupLoadMap* pPopupLoadMap, PopupSaveMap* pPopupSaveMap,
```

```

        PopupMapSize* pPopupMapSize)
void unimplemented()
void quit()
void set_title(char *new_title)

```

output variables

```

Xv_opaque frame

```

output

```

XView main window
XView Notice

```

class access programs

```

static Menu_item cfMenuFileLoad(Menu_item, Menu_generate)
static Menu_item cfMenuFileSave(Menu_item, Menu_generate)
static Menu_item cfMenuFileQuit(Menu_item, Menu_generate)
static Menu_item cfMenuMapRedraw(Menu_item, Menu_generate)
static Menu_item cfMenuMapClear(Menu_item, Menu_generate)
static Menu_item cfMenuMapChangeSize(Menu_item, Menu_generate)
static Menu_item cfMenuUnimplemented(Menu_item, Menu_generate)
static void cfRepaint(Canvas, Xv_window, Display, Window, Xv_xrectlist)
static void cfDestroy(Xv_opaque, Destroy_status)

```

class output variables

```

Notify_value notify_value

```

input variables

```

Attr_attribute Main::INSTANCE;
xv_get variable XV_KEY_DATA

```

external access

```

void PopupMapLoad::show()
void PopupMapSave::show()
void PopupMapSize::show()
void Map::init_draw(Display *display, Window xid, WinMap* pWinMap)
void Map::clear()
void Utils::cfNotice_OK()

```

transition

```

Si = init(o, p1, p2) -->
    create WinMap window with owner o. call p1->init_draw(display,xid,this)
Si = unimplemented() -->
    Utils::cfNotice_OK(f, "This function has not been implemented.")
    where f is frame created from Sj, where (∃Sj | (j < i) ∧ (Sj = init(o,p)))
Si = quit() -->
    call xv_destroy_safe(frame created from Sj), where (∃Sj | (j < i) ∧ (Sj =
    init(o,p)))
Si = set_title(new_title) --> set window title to "WestWorld -- <new_title>"
Si = cfMenuFileLoad(item, op) -->
    call PopupLoadMap* p->show() where p = xv_get(item, XV_KEY_DATA,
    INSTANCE)
Si = cfMenuFileSave(item, op) -->
    call PopupSaveMap* p->show() where p = xv_get(item, XV_KEY_DATA,
    INSTANCE)
Si = cfMenuFileQuit(item, op) -->

```

```

        call WinMap* p->quit() where p = xv_get(item, XV_KEY_DATA,
        INSTANCE)
Sj = cfMenuMapRedraw(item, op) -->
        call Map* p->draw() where p = xv_get(item, XV_KEY_DATA, INSTANCE)
Sj = cfMenuMapClear(item, op) -->
        call Map* p->clear() where p = xv_get(item, XV_KEY_DATA, INSTANCE)
Sj = cfMenuMapChangeSize(item, op) -->
        call PopupMapSize* p->show() where p = xv_get(item, XV_KEY_DATA,
        INSTANCE)
Sj = cfMenuUnimplemented(item, op) -->
        call WinMap* p->unimplemented() where p = xv_get(item, XV_KEY_DATA,
        INSTANCE)
Sj = cfRepaint(c, pw, d, w, x) -->
        call Map* p->draw() where p = xv_get(pw, XV_KEY_DATA, INSTANCE)
Si = cfDestroy(client, status) -->
        _Destroy(client, status)

```

end_BB

NOTES:

(1) assumption is made that init() comes before any other calls

define_BB Utils

class access programs

```

static void cfNotice_OK(Xv_opaque owner, char *message)
static int cfNotice_YN(Xv_opaque owner, char *message)

```

transition

```

cfNotice_OK(o, m) -->
    display XView notice with owner o, message, and Confirm button; wait until
    Confirm is pressed.
cfNotice_YN(o, m) -->
    display XView notice with owner o, message, and Yes/No buttons; wait until
    button is pressed, return TRUE (1) if Yes, FALSE (0) if No.

```

end_BB

III. TAM Specifications for Classes

CLASS: MAIN

TYPE IMPLEMENTED: <Main>

(1) SYNTAX

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4
<Invocation>					
main()	<void>	<int> argc	<char **> argv		
<Exit>					

CLASS OUTPUT VARIABLES

Variable Name	Type	Access
INSTANCE	<Attr_attribute>	publicly accessible

INPUT VARIABLES

Variable Name	Type	Access
WinMap::frame	<Xv_opaque>	direct

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3
Map::Map()	(construct.)			
Map::~~Map()	(destruct.)			
PopupLoadMap::init()	<void>	<Xv_opaque>	<Map*>	
PopupSaveMap::init()	<void>	<Xv_opaque>	<Map*>	
PopupMapSize::init()	<void>	<Xv_opaque>	<Map*>	
WinMap::init()	<void>	<Xv_opaque>	<Map*>	<PopupLoadMap*>
		#4 <PopupSaveMap*>	#5 <PopupMapSize*>	

(2) CANONICAL TRACES

$\text{canonical}(T_C) \leftrightarrow (T_C = \langle \text{Invocation} \rangle) \vee (T_C = \langle \text{Invocation} \rangle.\text{main}())$

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for <Invocation> and main().

$T_C.\langle \text{Invocation} \rangle \equiv \langle \text{Invocation} \rangle;$

ADD-TO-TRACE(T_m , Map()) where T_m is trace for Map object created by <Invocation>

Completeness (3): The predicates in the LIIC of each table partition the intended domain of the relation:

- no partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by LHC.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- $\langle \text{Invocation} \rangle$ is canonical.

Consistency (3): All RHC values are unique:

- One value.

$T_C.\text{main}() \equiv$

conditions	equivalences
$T_C = \langle \text{Invocation} \rangle$	$\text{main}();$ $\text{ADD-TO-TRACE}(T_{wm}, \text{init}(\text{NULL}, p_m, p_{plm}, p_{psm}, p_{pms}));$ $\text{ADD-TO-TRACE}(T_{plm}, \text{init}(p_{wm} \rightarrow \text{frame}, p_m));$ $\text{ADD-TO-TRACE}(T_{psm}, \text{init}(p_{wm} \rightarrow \text{frame}, p_m));$ $\text{ADD-TO-TRACE}(T_{pms}, \text{init}(p_{wm} \rightarrow \text{frame}, p_m));$ where T_{wm} is trace for WinMap object created by $\text{main}()$ and p_{wm} is pointer to that object, T_{plm} is a trace for PopupLoadMap object created by $\text{main}()$ and p_{plm} is pointer to that object, T_{psm} is a trace for PopupSaveMap object created by $\text{main}()$ and p_{psm} is pointer to that object, T_{pms} is a trace for PopupMapSize object created by $\text{main}()$ and p_{pms} is pointer to that object, and p_m is pointer to Map object created by $\langle \text{Invocation} \rangle$
else	$\% \text{main already called} \%$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- event defined by LHC; traces and pointers used in RHC are specified by event in T_C [$\langle \text{Invocation} \rangle$] or the current event [$\text{main}()$]

Consistency (2): All traces specified in the RHIC of the equivalence section are canonical:

- $\text{main}()$ is canonical.

Consistency (3): All RHC values are unique:

- One value, one error.

$T_C.\langle \text{Exit} \rangle \equiv T_C;$

$\text{ADD-TO-TRACE}(T_m, \sim \text{Map}())$ where T_m is trace for Map object created by $\langle \text{Invocation} \rangle$

Completeness (3): The predicates in the LHIC of each table partition the intended domain of the relation:

- no partition.

Completeness (4): The predicates in the RHIC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by LHIC.

Consistency (2): All traces specified in the RHIC of the equivalence section are canonical:

- T_C is canonical by definition.

Consistency (3): All RHC values are unique:

- One value.

(4) VALUES

OUTPUT VALUES

V[INSTANCE](T) =

conditions	values
T = <Invocation>	%undefined%
T = main()	xv_unique_key()

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Since T is canonical, the conditions partition the canonical trace and therefore give a full partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- N/A

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- One value, one error.

RETURN VALUES

<none>

CLASS: MAP

TYPE IMPLEMENTED: <Map>

(1) SYNTAX

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4
Map	(constructor)				
~Map	(destructor)				
init_draw	<void>	<Display *> display	<Window> xid	<WinMap*> pWinMap	
draw	<void>				
clear	<void>				
change_size	<int> change_ok	<Xv_opaque> frame	<double> new_width	<double> new_length	<int> new_scale
load	<int> loadsave_ok	<Xv_opaque> frame	char* loadfile		
save	<int> loadsave_ok	<Xv_opaque> frame	char* savefile		
loadline	<int> loadline_ok	<Xv_opaque> frame	<int> lineno	<char*> line	

OUTPUT VARIABLES

Variable Name	Type	Access
width	<double>	public
length	<double>	public
scale	<int>	public
filename	<char*>	public
change_ok	<int>	function return
loadsave_ok	<int>	function return
loadline_ok	<int>	function return
(output screen)	(X display window)	N/A

INPUT VARIABLES

Variable Name	Type	Access
file_status ¹	pseudo	input pseudo-event

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4
Utils::cfNotice_OK	<void>	<Xv_opaque> owner	<char *> message		
Utils::cfNotice_YN	<int> yn_answer	<Xv_opaque> owner	<char *> message		
WinMap::set_title	<void>	<char*> new_title			
fopen	<FILE*> stream	<char*> filename	<char*> mode		

¹This variable does not necessarily exist, rather it is a placemaker for the results from calls to the filesystem.

fclose	<int> err	<FILE*> stream			
fgets	<int> err	<char*> s	<int> n	<FILE*> stream	
fputs	<int> err	<char*> s	<FILE*> stream		

(2) CANONICAL TRACES

$\text{canonical}(T) \leftrightarrow$
 $(T = \text{Map}()) \vee$
 $(T = \text{Map}().\text{init_draw}(d, xw, wf).$
 $[\text{change_size}(f, w, l, s)]_{i=0}^1. [\text{change_size}(f, w', l', s')] \wedge \text{bad_values}(w', l', s')]_{i=0}^1.$
 $[\text{load}(fr, fi) \vee \text{save}(fr, fi)]_{i=0}^1. [\text{file_status}]_{i=0}^1.$
 $((\text{loadline}(f, i, l) \wedge \text{bad_line}(l)) \vee [\text{loadline}(f, i, l_i) \wedge \text{not}(\text{bad_line}(l_i))]_{i=0}^m)$

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

AUXILIARY FUNCTIONS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4	Arg#5
bad_values	<boolean>	<double> new_width	<double> new_length	<int> new_scale		
bad_line	<boolean>	<char*> line				
parse	<boolean>	<trace>	<trace>	<trace>	<trace>	

bad_values(w,l,s) =

conditions	equivalences
$(w < 1) \vee (l > 1) \vee$ $(s < 1) \vee (s > 100)$	true
else	false

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

bad_line(l) =

conditions	equivalences
l is of form "box <locx> <locy> <width> <length> [<height>] ∨ l is of form "cyl[<inder>] <locx> <locy> <radius> [<height>] ∨ l[0] = '#' ∨ l[0] = '\0'	false
else	true

Completeness (3): The predicates in the LHIC of each table partition the intended domain of the relation:

- Else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

parse(S,S1,S2,S3,S4,S5,S6) =

conditions	equivalences
$(S = S1.S2.S3.S4) \wedge$ $(S1 = \text{Map}[\text{init_draw}(d,xw,wf)]_{i=0}^1) \wedge$ $(S2 = [\text{change_size}(f,w,l,s) \wedge \text{not}(\text{bad_values}(w,l,s))]_{i=0}^1) \wedge$ $(S3 = [\text{change_size}(f',w',l',s') \wedge \text{bad_values}(w',l',s')]_{i=0}^1) \wedge$ $(S4 = [\text{load}(fr,fi) \vee \text{save}(fr,fi)]_{i=0}^1) \wedge$ $(S5 = [\text{file_status}]_{i=0}^1) \wedge$ $(S6 = (\text{loadline}(f,i,l) \wedge \text{bad_line}(l)) \vee$ $[\text{loadline}(f,i,l_i) \wedge \text{not}(\text{bad_line}(l_i))]_{i=0}^m)$	true
else	false

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for access programs Map, ~Map(), init_draw, draw, clear, change_size, load, save, and loadline as well as input event file_status.

T.Map() $\equiv$ Map()

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- No partitioning of domain, therefore complete

Consistency (2): All traces specified in the RIIC of the equivalence section are canonical:

- Map() is a canonical trace

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

T.~Map() $\equiv$ T²

²The destructor ~Map() will probably result in state changes for the object, but since it is about to disappear from scope, its effect on the trace does not matter since following ~Map(), the object is undefined.

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- No partitioning of domain, therefore complete

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- Map() is a canonical trace

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

$T.init_draw(d, xw, wf) \equiv Map().init_draw(d, xw, wf).C.CE.FN.FS.L$, where $parse(T, I, C, CE, FN, FS, L)$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- The predicates in RHC are comprised of canonical trace elements from LHC or the stimulus itself, and are therefore all defined.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- The trace given is canonical.

Consistency (3): All RHC values are unique:

- Only one value.

$T.draw() \equiv$

conditions	equivalences
$T = Map()$	%uninitialized%
else	T

Completeness (3): The predicates in the LHIC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T is defined by other side of equivalence.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T is canonical by definition.

Consistency (3): All RHC values are unique:

- One value, one error.

$T.clear() \equiv$

conditions	equivalences
$T = Map()$	%uninitialized%
else	equiv = I.C; ADD-TO-TRACE(T_{wf} , set_title("<NONE>")); where $parse(T, I, C, CE, FN, FS, L)$ and $I = init_draw(d, xw, wf)$

Completeness (3): The predicates in the LHIC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHIC are defined whenever the corresponding predicate in the LHIC is 'true':

- T is defined by LHS.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- Map().init_draw() and Map().init_draw().change_size() are canonical

Consistency (3): All RHC values are unique:

- One value, one error.

$T.change_size(f,w,l,s) \equiv$
conditions

equivalences

$T = Map()$	%uninitialized%
$(w < 1) \vee (l < 1)$	equiv = I.C.change_size(f,w,l,s).FN.FS.L; ADD-TO-TRACE(T_U , cfNotice_OK(f, "Width and Length must be at least 1.0m.")) where parse(T, I, C, CE, FN, FS, L) and T_U is the class access trace for Utils
$(w \geq 1) \wedge (l \geq 1) \wedge ((s < 1) \vee (s > 100))$	equiv = I.C.change_size(f,w,l,s).FN.FS.L; ADD-TO-TRACE(T_U , cfNotice_OK(f, "Scale must be in the range of 1 to 100.")) where parse(T, I, C, CE, FN, FS, L) and T_U is the class access trace for Utils
else	I.change_size(f,w,l,s).FN.FS.L where parse(T, I, C, CE, FN, FS, L)

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first case has only constructor, others assume init_draw() in trace; second and third separated by w/l comparisons, else insures full partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- RHC items defined by call and parsed trace.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- The traces shown are all canonical.

Consistency (3): All RHC values are unique:

- Second and third cases are different by the cfNotice_OK calls; third replaces any error present.

$T.load(fr,fi) \equiv$

conditions

equivalences

$T = Map()$	%uninitialized%
$fopen(fi, "r") = NULL$	equiv = I.C.(file_status = FILE_OPEN_ERR); ADD-TO-TRACE(this, clear()); ADD-TO-TRACE(T_f , fopen(fi, "r")); ADD-TO-TRACE(T_U , cfNotice_OK(f, "The file could not be opened.")); where parse(T, I, C, CE, FN, FS, L), T_U is the class access trace for Utils, and T_f is the trace for the file system
$(fopen(fi, "r") \neq NULL) \wedge (loadline(fr, i, s_i) = FALSE)$	equiv = I.C; ADD-TO-TRACE(this, clear()); ADD-TO-TRACE(this, [loadline(fr, i, s_i)] _{i=0} ⁿ); ADD-TO-TRACE(T_f , F=fopen(fi, "r"), [fgets(F, N, s_i)] _{i=0} ⁿ , fclose(F)); where parse(T, I, C, CE, FN, FS, L) and T_f is the trace for the file system
$(F=fopen(fi, "r") \neq NULL) \wedge (loadline(fr, i, s_i) = TRUE) \wedge (fgets(F,N,s_i) = NULL) \wedge (notfeof(F))$	equiv = I.C; ADD-TO-TRACE(this, clear()); ADD-TO-TRACE(this, [loadline(fr, i, s_i)] _{i=0} ⁿ); ADD-TO-TRACE(T_f , F=fopen(fi, "r"), [fgets(F, N, s_i)] _{i=0} ⁿ , fclose(F)); ADD-TO-TRACE(T_U , cfNotice_OK(f, "An error occurred reading the file.")); where parse(T, I, C, CE, FN, FS, L) T_U is the class access trace for Utils, and T_f is the trace for the file system

$(F = \text{fopen}(fi, "r") \neq \text{NULL}) \wedge$ $(\text{loadline}(fr, i, si) = \text{TRUE}) \wedge$ $(\text{fclose}(F) \neq 0)$	$\text{equiv} = \text{I.C.}(\text{file_status} = \text{FILE_CLOSE_ERR});$ $\text{ADD-TO-TRACE}(\text{this}, \text{clear}());$ $\text{ADD-TO-TRACE}(\text{this}, [\text{loadline}(fr, i, si)]_{i=0}^n);$ $\text{ADD-TO-TRACE}(T_f, F = \text{fopen}(fi, "r"), [\text{fgets}(fi, N, si)]_{i=0}^n, \text{fclose}(F));$ $\text{ADD-TO-TRACE}(T_u, \text{cfNotice_OK}(f, "An error occurred closing the file."));$ where $\text{parse}(T, I, C, CE, FN, FS, L)$, T_u is the class access trace for Utils, and T_f is the trace for the file system
else	$\text{equiv} = \text{I.C.}(\text{load}(fr, fi).[\text{loadline}(fr, i, si)]_{i=0}^n);$ $\text{ADD-TO-TRACE}(\text{this}, \text{clear}());$ $\text{ADD-TO-TRACE}(\text{this}, [\text{loadline}(fr, i, si)]_{i=0}^n);$ $\text{ADD-TO-TRACE}(T_f, F = \text{fopen}(fi, "r"), [\text{fgets}(fi, N, si)]_{i=0}^n, \text{fclose}(F));$ $\text{ADD-TO-TRACE}(T_{wf}, \text{set_title}(fi));$ where $\text{parse}(T, I, C, CE, FN, FS, L)$, $I = \text{init_draw}(d, xw, wf)$, and T_f is the trace for the file system

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T is defined by LHS.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- $\text{I.C.}(\text{filestatus})$, $\text{I.C.}(\text{I.C.}(\text{load}.\text{loadline}^*))$ are all canonical

Consistency (3): All RHC values are unique:

- Each different in either output trace or modifications to other traces.

$T.\text{save}(fr, fi) \equiv$

conditions	equivalences
$T = \text{Map}()$	$\%uninitialized\%$
$\text{fopen}(fi, "r") \neq \text{NULL} \wedge$ $\text{cfNoticeYN}(fr, "File exists. Overwrite it?") = \text{FALSE}$	$\text{equiv} = \text{I.C.}(\text{file_status} = \text{FILE_OPEN_ERR}).L;$ $\text{ADD-TO-TRACE}(T_f, F = \text{fopen}(fi, "r").\text{fclose}(F));$ $\text{ADD-TO-TRACE}(T_u, \text{cfNotice_YN}(fr, "File exists. Overwrite it?"));$ where $\text{parse}(T, I, C, CE, FN, FS, L)$, T_u is the class access trace for Utils, and T_f is the trace for the file system
$(F = \text{fopen}(fi, "r") \neq \text{NULL} \wedge$ $\text{cfNoticeYN}(fr, "File exists. Overwrite it?") = \text{TRUE}) \wedge$ $(F2 = \text{fopen}(fi, "w") = \text{NULL})$	$\text{equiv} = \text{I.C.}(\text{file_status} = \text{FILE_OPEN_ERR}).L;$ $\text{ADD-TO-TRACE}(\text{this}, \text{clear}());$ $\text{ADD-TO-TRACE}(T_f, \text{fopen}(fi, "r"));$ $\text{ADD-TO-TRACE}(T_u, \text{cfNotice_YN}(fr, "File exists. Overwrite it?"),$ $\text{cfNotice_OK}(f, "The file could not be opened."));$ where $\text{parse}(T, I, C, CE, FN, FS, L)$, T_u is the class access trace for Utils, and T_f is the trace for the file system
$(F = \text{fopen}(fi, "r") \neq \text{NULL} \wedge$ $\text{cfNoticeYN}(fr, "File exists. Overwrite it?") = \text{TRUE}) \wedge$ $(F2 = \text{fopen}(fi, "w") \neq \text{NULL})$ $\wedge \text{fputs}(F2, si) \neq 0$	$\text{equiv} = \text{I.C.}(\text{file_status} = \text{FILE_WRITE_ERR}).L;$ $\text{ADD-TO-TRACE}(T_f, F = \text{fopen}(fi, "r"), \text{fclose}(F), F2 = \text{fopen}(fi, "w")$ $[\text{fputs}(F2, si)]_{i=0}^n, \text{fclose}(F2));$ $\text{ADD-TO-TRACE}(T_u, \text{cfNotice_YN}(fr, "File exists. Overwrite it?"),$ $\text{cfNotice_OK}(f, "An error occurred writing the file."));$ where $\text{parse}(T, I, C, CE, FN, FS, L)$, T_u is the class access trace for Utils, and T_f is the trace for the file system

$(F=fopen(fi, "r") \neq NULL \wedge$ $cfNoticeYN(fr, "File exists. Overwrite it?") = TRUE) \wedge$ $(F2=fopen(fi, "w") \neq NULL) \wedge$ $fputs(F2, s_i) \neq 0 \wedge$ $fclose(F2) \neq 0$	$equiv = I.C.(file_status = FILE_CLOSE_ERR).L;$ $ADD-TO-TRACE(T_f, T_f, F=fopen(fi, "r"), fclose(F), F2=fopen(fi, "w") [fputs(F2, s_i)]_{i=0}^n, fclose(F2));$ $ADD-TO-TRACE(T_u, cfNotice_YN(fr, "File exists. Overwrite it?"), cfNotice_OK(f, "An error occurred closing the file."));$ where $parse(T, I, C, CE, FN, FS, L)$, T_u is the class access trace for Utils, and T_f is the trace for the file system
$(F=fopen(fi, "r") \neq NULL \wedge$ $cfNoticeYN(fr, "File exists. Overwrite it?") = TRUE) \wedge$ $(F2=fopen(fi, "w") \neq NULL) \wedge$ $fputs(F2, s_i) \neq 0 \wedge$ $fclose(F2) = 0$	$equiv = I.C.save(fr, fi).L$ $ADD-TO-TRACE(this, clear());$ $ADD-TO-TRACE(T_u, cfNotice_YN(fr, "File exists. Overwrite it?"));$ $ADD-TO-TRACE(T_f, F=fopen(fi, "r"), fclose(F), F2=fopen(fi, "w") [fputs(F2, s_i)]_{i=0}^n, fclose(F2));$ $ADD-TO_TRACE(T_{wf}, set_title(fi));$ where $parse(T, I, C, CE, FN, FS, L)$, $I=init_draw(d, xw, wf)$, T_u is the class access trace for Utils, and T_f is the trace for the file system
$(F=fopen(fi, "r") = NULL) \wedge$ $(F2=fopen(fi, "w") = NULL)$	$equiv = I.C.(file_status = FILE_OPEN_ERR).L;$ $ADD-TO-TRACE(this, clear());$ $ADD-TO-TRACE(T_f, fopen(fi, "r"));$ $ADD-TO-TRACE(T_u, cfNotice_OK(f, "The file could not be opened."));$ where $parse(T, I, C, CE, FN, FS, L)$, T_u is the class access trace for Utils, and T_f is the trace for the file system
$(F=fopen(fi, "r") = NULL) \wedge$ $(F2=fopen(fi, "w") \neq NULL) \wedge$ $fputs(F2, s_i) \neq 0$	$equiv = I.C.(file_status = FILE_WRITE_ERR).L;$ $ADD-TO-TRACE(T_f, F=fopen(fi, "r"), fclose(F), F2=fopen(fi, "w") [fputs(F2, s_i)]_{i=0}^n, fclose(F2));$ $ADD-TO-TRACE(T_u, cfNotice_OK(f, "An error occurred writing the file."));$ where $parse(T, I, C, CE, FN, FS, L)$, T_u is the class access trace for Utils, and T_f is the trace for the file system
$(F=fopen(fi, "r") = NULL) \wedge$ $(F2=fopen(fi, "w") \neq NULL) \wedge$ $fputs(F2, s_i) \neq 0 \wedge$ $fclose(F2) \neq 0$	$equiv = I.C.(file_status = FILE_CLOSE_ERR).L;$ $ADD-TO-TRACE(T_f, T_f, F=fopen(fi, "r"), fclose(F), F2=fopen(fi, "w") [fputs(F2, s_i)]_{i=0}^n, fclose(F2));$ $ADD-TO-TRACE(T_u, cfNotice_OK(f, "An error occurred closing the file."));$ where $parse(T, I, C, CE, FN, FS, L)$, T_u is the class access trace for Utils, and T_f is the trace for the file system
else	$equiv = I.C.save(fr, fi).L$ $ADD-TO-TRACE(this, clear());$ $ADD-TO-TRACE(T_f, F=fopen(fi, "r"), fclose(F), F2=fopen(fi, "w") [fputs(F2, s_i)]_{i=0}^n, fclose(F2));$ $ADD-TO_TRACE(T_{wf}, set_title(fi));$ where $parse(T, I, C, CE, FN, FS, L)$, $I=init_draw(d, xw, wf)$, and T_f is the trace for the file system

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T is defined by LHS.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- $I.C.filestatus$, $I.C.I.C.save.L$ are all canonical

Consistency (3): All RHC values are unique:

- Each different in either output trace or modifications to other traces.

$T.loadline(fr, n, l) \equiv$

conditions	equivalences
$l[0] = \# \vee l[0] = \backslash 0'$	T
$strcmp(l, "box", 3) = 0 \wedge$ l not in proper box format	$equiv = I.C.FN.FS.loadline()$ $ADD-TO-TRACE(T_u, cfNotice_OK(fr,$ "Map file format error: bad box @ line <n>")); where $parse(T, I, C, CE, FN, FS, L)$
$strcmp(l, "cyl", 3) = 0 \wedge$ l not in proper cylinder format	$equiv = I.C.FN.FS.loadline()$ $ADD-TO-TRACE(T_u, cfNotice_OK(fr,$ "Map file format error: bad cylinder @ line <n>")); where $parse(T, I, C, CE, FN, FS, L)$
$l[0] \neq \# \wedge l[0] \neq \backslash 0' \wedge strcmp(l, "box", 3)$ $\neq 0 \wedge strcmp(l, "cyl", 3) \neq 0 \wedge$ l not in proper cylinder format	$equiv = I.C.FN.FS.loadline()$ $ADD-TO-TRACE(T_u, cfNotice_OK(fr,$ "Map file format error: unknown object @ line <n>")); where $parse(T, I, C, CE, FN, FS, L)$
else	$equiv = I.C.FN.FS.L.loadline()$ where $parse(T, I, C, CE, FN, FS, L)$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition, tests on l all different

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T is defined by LHS.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- $I.C.FN.FS.loadline^*$ is canonical

Consistency (3): All RHC values are unique:

- One value, one error.

$T.file_status() \equiv$

conditions	equivalences
$T = Map()$	%uninitialized%
else	$equiv = I.C.CE.FN.file_status.L;$ where $parse(T, I, C, CE, FN, FS, L)$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T is defined by LHS.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- $I.C.CE.FN.file_status.L$ is canonical

Consistency (3): All RHC values are unique:

- One value, one error.

(4) VALUES

OUTPUT VALUES

V[width](T) =

conditions	values
parse(T, I, C, CE, FN, FS, L) $\wedge$ C $\neq$ _	w where C = change_size(f,w,l,s)
else	12

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- first case is defined since change_size() must be defined if C $\neq$ _; second case is constant.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- The first case value may be the same as the constant, but not always, requiring partitioning.

V[length](T) =

conditions	values
parse(T, I, C, CE, FN, FS, L) $\wedge$ C $\neq$ _	l where C = change_size(f,w,l,s)
else	12

Consistency/Completeness: Same as above.

V[scale](T) =

conditions	values
parse(T, I, C, CE, FN, FS, L) $\wedge$ C $\neq$ _	s where C = change_size(f,w,l,s)
else	40

Consistency/Completeness: Same as above.

V[filename](T) =

conditions	values
parse(T, I, C, CE, FN, FS, L) $\wedge$ FN $\neq$ _	fi where FN = load(fr, fi) $\vee$ FN = save(fr, fi)
else	""

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- first case is defined since load() or save() must be in trace if FN $\neq$ _; second case is constant.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- One has real filename, other is blank.

V[change_ok](T) =

conditions	values
$\text{parse}(T, I, C, CE, FN, FS, L) \wedge C = _ \wedge CE = _$	%undefined%
$\text{parse}(T, I, C, CE, FN, FS, L) \wedge C \neq _ \wedge CE = _$	1
else	0

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- cases one and two are distinguished by C test; else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- all outputs are constant or error and therefore defined.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- true.

$V[\text{loadsave_ok}](T) =$

conditions	values
$\text{parse}(T, I, C, CE, FN, FS, L) \wedge FS = \text{FILE_OPEN_ERR} \wedge FS = \text{FILE_WRITE_ERR} \wedge FS = \text{FILE_CLOSE_ERR}$	0
$\text{parse}(T, I, C, CE, FN, FS, L) \wedge FN \neq _$	1
else	%undefined%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- logic in load() + save() equivalences prohibits having a value in FS when value in FN, therefore these conditions are separate; else insures partition for other cases.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- all outputs are constant or error and therefore defined.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- true.

$V[\text{loadline_ok}](T) =$

conditions	values
$\text{parse}(T, I, C, CE, FN, FS, L) \wedge L = \text{loadline}(f, i, l) \wedge \text{bad_line}(l)$	0
$\text{parse}(T, I, C, CE, FN, FS, L) \wedge L = [\text{loadline}(f, i, l)]_{i=1}^m \wedge \text{not}(\text{bad_line}(l_1))$	1
else	%undefined%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Either L is defined and bad or it is defined and ok; else insures partition for other cases.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- all outputs are constant or error and therefore defined.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- true.

$V[(output_screen)](T) =$

conditions	values
$T = Map()$	%no_output%
$parse(T, I, C, CE, FN, FS, L) \wedge$ $I = Map().init_draw(d, xw, wf) \wedge$ $C = _$	rect of size 12*40 x 12*40 with objects defined by L (if not bad_line) drawn in window with Display *d, Window xw
$parse(T, I, C, CE, FN, FS, L) \wedge$ $I = Map().init_draw(d, xw, wf) \wedge$ $C = change_size(f, w, l, s)$	rect of size w*s x l*s with objects defined by L (if not bad_line) drawn in window with Display *d, Window xw

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- If trace does not have just Map(), then I will be equal to Map().init_draw combination, and C comparison insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- values are either constant or defined from variables present in LHC, therefore defined.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- case 3 may be same as constant values in case two, but not always, requiring partitioning.

RETURN VALUES

Program Name	Argument No	Value
change_size	Value	change_ok
load	Value	loadsave_ok
save	Value	loadsave_ok
loadline	Value	loadline_ok

Completeness (2): There is one output function/relation that specifies each output value:

- There are output values defined above for each value in the table: V[change_ok], V[loadsave_ok], and V[loadline_ok]; the other values above are not return values.

CLASS: POPUPLOADMAP

TYPE IMPLEMENTED: <PopupLoadMap>

(1) SYNTAX

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
init	<void>	<Xv_opaque> owner_frame	<Map*> pMap
show	<void>		
load	<void>	<Panel_item> item	

OUTPUT VARIABLES

Variable Name	Type	Access
(popup window)	(XView Popup window)	N/A

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
cfLoad	<void>	<Panel_item>	<Event>

INPUT VARIABLES

Variable Name	Type	Access
load_error	<int>	input pseudo-event
Map::filename	<char*>	direct access
entered_filename	<int>	XView xv_get value
FRAME_CMD_PUSHPIN_IN	<int>	XView xv_get value
XV_KEY_DATA	<Xv_opaque>	XView xv_get value
INSTANCE	<Attr_attribute>	direct access

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4
Map::load	<int> load_error	<Xv_opaque> popup_frame	<char*> loadfile		

(2) CANONICAL TRACES

canonical(T_i) $\leftrightarrow$ ($T_i = _$) $\vee$ ($T_i = \text{init}(o,p)$) $\vee$ ($T_i = \text{init}(o,p).\text{show}()$) $\vee$ ($T_i = \text{init}(o,p).\text{show}().\text{load}(it)$) $\vee$ ($T_i = \text{init}(o,p).\text{show}().\text{load}(it).\text{load_error}$)

canonical(T_c) $\leftrightarrow$ ($T_c = _$)

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

AUXILIARY FUNCTIONS

parse(S,S1,S2,S3,S4) =

conditions	equivalences
$(S = S1.S2.S3.S4) \wedge$ $(S1 = [init(o,p)]_{i=0}^1) \wedge$ $(S2 = [show()]_{i=0}^1) \wedge$ $(S3 = [load(it)]_{i=0}^1) \wedge$ $(S4 = [load_error]_{i=0}^1)$	true
else	false

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for init, show, load, load_error, and cfLoad

T.init(o,p) =

conditions	equivalences
T = _	init(o,p)
T ≠ _	%already_initialized%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- If one LHC condition is true, the other must be false, and they therefore partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- init(o,p) is defined by event itself, other RHC item is error message.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- Only one is specified, init(), and it is canonical.

Consistency (3): All RHC values are unique:

- One is value, one is error.

T.show() =

conditions	equivalences
T = _	%uninitialized%
else	I.show() where parse(T, I, S, L, LE)

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- First case is empty trace, second has something in trace, third is else. insuring partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- First two cases are errors, in last I must be defined since T is not empty.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- Trace init().show() [I.show()] is in the canonical trace.

Consistency (3): All RHC values are unique:

- True.

T.load(it) $\equiv$

conditions	equivalences
T = _	%uninitialized%
T = init(o,p)	%undisplayed%
parse(T, I, S, L, LE) $\wedge$ S $\neq$ _ $\wedge$ I=init(o,p) $\wedge$ p->load() = TRUE	equivalence = I.S.load(it); ADD-TO-TRACE(T _p , load(f, entered_filename)) where f is frame created by init()
else	equivalence = I.S.load(it).load_error; ADD-TO-TRACE(T _p , load(f, entered_filename)) where parse(T, I, S, L, LE) $\wedge$ I=init(o,p) $\wedge$ load_error = load() $\wedge$ f is frame created by init()

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- First two are obviously different, third has show() in T, else separates third from fourth.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- I and S defined for traces in 3rd/4th cases; p, load_error defined as given; entered_filename and f defined if popup has been created (since init must be in trace, that is true).

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- init().show().load() and init().show().load().load_error are canonical

Consistency (3): All RHC values are unique:

- 3rd + 4th cases differ in equivalence

T.load_error $\equiv$

conditions	equivalences
T = init(o,p).show().load(it)	T.load_error
else	%undefined%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T defined by LHC.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T.load_error canonical if T is as defined by LHC

Consistency (3): All RHC values are unique:

- One value, one error.

T_C.cfLoad(item,e) $\equiv$ T_C; ADD-TO-TRACE(T_p, load(item))
where PopLoadMap* p = xv_get(item, XV_KEY_DATA, INSTANCE);

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- If load occurs, the PopLoadMap object must have already been created, and p will be valid.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T_C canonical by definition.

Consistency (3): All RHC values are unique:

- Only one value.

(4) VALUES

OUTPUT VALUES

V[popup_frame](T) =

conditions	values
T = _	%undefined%
else	frame created via init function

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- frame is defined by init, which must be part of any non-empty trace.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- Either frame or error.

V[(popup_window)](T) =

conditions	values
T = _	%undefined%
T = init(o,p)	%undisplayed%
T = init(o,p).show()	popup window displayed on screen; Filename field = p->filename; value may be modified by user
T = init(o,p).show().load(it) ^ xv_get(frame created by init(), FRAME_CMD_PUSHPIN_IN) = TRUE	popup field set to value from p-> as given above
T = T1.load(it) ^ xv_get(frame created by init(), FRAME_CMD_PUSHPIN_IN) = FALSE	popup window disappears from screen
else	popup forced to remain on screen, with values as modified by user

WestWorld: Load Map

Filename:

<string>

Load

[PopupLoadMap]

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- LHC partitions the entire canonical trace.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- window and fields are created by init, which is included in RHC trace

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- Either has constant (default) appearance or one modified by user input.

RETURN VALUES

(none)

CLASS: POPUPSAVEMAP

TYPE IMPLEMENTED: <PopupSaveMap>

(1) SYNTAX

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
init	<void>	<Xv_opaque> owner_frame	<Map*> pMap
show	<void>		
save	<void>	<Panel_item> item	

OUTPUT VARIABLES

Variable Name	Type	Access
(popup window)	(XView Popup window)	N/A

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
cfSave	<void>	<Panel_item>	<Event>

INPUT VARIABLES

Variable Name	Type	Access
save_error	<int>	input pseudo-event
Map::filename	<char*>	direct access
entered_filename	<int>	XView xv_get value
FRAME_CMD_PUSHPIN_IN	<int>	XView xv_get value
XV_KEY_DATA	<Xv_opaque>	XView xv_get value
INSTANCE	<Attr_attribute>	direct access

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4
Map::save	<int> save_error	<Xv_opaque> popup_frame	<char*> savefile		

(2) CANONICAL TRACES

canonical(T_i) $\leftrightarrow$ ($T_i = _$) $\vee$ ($T_i = \text{init}(o,p)$) $\vee$ ($T_i = \text{init}(o,p).\text{show}()$) $\vee$ ($T_i = \text{init}(o,p).\text{show}().\text{save}(it)$) $\vee$ ($T_i = \text{init}(o,p).\text{show}().\text{save}(it).\text{save_error}$)

canonical(T_c) $\leftrightarrow$ ($T_c = _$)

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

AUXILIARY FUNCTIONS

parse(S,S1,S2,S3,S4) =

conditions	equivalences
$(S = S1.S2.S3.S4) \wedge$ $(S1 = [init(o,p)]_{i=0}^1) \wedge$ $(S2 = [show()]_{i=0}^1) \wedge$ $(S3 = [save(it)]_{i=0}^1) \wedge$ $(S4 = [save_error]_{i=0}^1)$	true
else	false

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for init, show, save, save_error, and cfSave

T.init(o,p) ≡

conditions	equivalences
T = _	init(o,p)
T ≠ _	%already_initialized%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- If one LHC condition is true, the other must be false, and they therefore partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- init(o,p) is defined by event itself, other RHC item is error message.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- Only one is specified, init(), and it is canonical.

Consistency (3): All RHC values are unique:

- One is value, one is error.

T.show() ≡

conditions	equivalences
T = _	%uninitialized%
else	I.show() where parse(T, I, S, L, LE)

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- First case is empty trace, second has something in trace, third is else, insuring partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- First two cases are errors, in last I must be defined since T is not empty.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- Trace init().show() [I.show()] is in the canonical trace.

Consistency (3): All RHC values are unique:

- True.

$T.save(it) \equiv$

conditions	equivalences
$T = _$	%uninitialized%
$T = init(o,p)$	%undisplayed%
$parse(T, I, S, L, LE) \wedge$ $S \neq _ \wedge I = init(o,p) \wedge$ $p \rightarrow save() = TRUE$	equivalence = I.S.save(it); ADD-TO-TRACE(T_p , save(f, entered_filename)) where f is frame created by init()
else	equivalence = I.S.save(it).save_error; ADD-TO-TRACE(T_p , save(f, entered_filename)) where $parse(T, I, S, L, LE) \wedge I = init(o,p) \wedge save_error = save() \wedge f$ is frame created by init()

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- First two are obviously different, third has show() in T, else separates third from fourth.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- I and S defined for traces in 3rd/4th cases; p, save_error defined as given; entered_filename and f defined if popup has been created (since init must be in trace, that is true).

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- init().show().save() and init().show().save().save_error are canonical

Consistency (3): All RHC values are unique:

- 3rd + 4th cases differ in equivalence

$T.save_error \equiv$

conditions	equivalences
$T = init(o,p).show().save(it)$	$T.save_error$
else	%undefined%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T defined by LHC.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T.save_error canonical if T is as defined by LHC

Consistency (3): All RHC values are unique:

- One value, one error.

$T_c.cfSave(item,e) \equiv T_c; ADD-TO-TRACE(T_p, save(item))$
where $PopSaveMap^* p = xv_get(item, XV_KEY_DATA, INSTANCE);$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- If save occurs, the PopupSaveMap object must have already been created, and p will be valid.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T_c canonical by definition.

Consistency (3): All RHC values are unique:

- Only one value.

(4) VALUES

OUTPUT VALUES

$V[\text{popup_frame}](T) =$

conditions	values
$T = _$	%undefined%
else	frame created via init function

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- frame is defined by init, which must be part of any non-empty trace.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- Either frame or error.

$V[(\text{popup_window})](T) =$

conditions	values
$T = _$	%undefined%
$T = \text{init}(o,p)$	%undisplayed%
$T = \text{init}(o,p).\text{show}()$	popup window displayed on screen; Filename field = p->filename; value may be modified by user
$T = \text{init}(o,p).\text{show}().\text{save}(it) \wedge$ $xv_get(\text{frame created by init}(),$ $\text{FRAME_CMD_PUSHPIN_IN}) = \text{TRUE}$	popup field set to value from p-> as given above
$T = T1.\text{save}(it) \wedge$ $xv_get(\text{frame created by init}(),$ $\text{FRAME_CMD_PUSHPIN_IN}) = \text{FALSE}$	popup window disappears from screen
else	popup forced to remain on screen, with values as modified by user

○
WestWorld: Save Map

Filename: _____<string>_____

Save

[PopupSaveMap]

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- LHC partitions the entire canonical trace.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- window and fields are created by init, which is included in RHC trace

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- Either has constant (default) appearance or one modified by user input.

RETURN VALUES

(none)

CLASS: POPUPMAPSIZE

TYPE IMPLEMENTED: <PopupMapSize>

(1) SYNTAX

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
init	<void>	<Xv_opaque> owner_frame	<Map*> pMap
show	<void>		
change	<void>	<Panel_item> item	

OUTPUT VARIABLES

Variable Name	Type	Access
(popup window)	(XView Popup window)	N/A

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
cfChange	<void>	<Panel_item>	<Event>

INPUT VARIABLES

Variable Name	Type	Access
change_error	<int>	input pseudo-event
Map::width	<double>	direct access
Map::length	<double>	direct access
Map::scale	<int>	direct access
entered_width	<char *>	XView xv_get value
entered_length	<char *>	XView xv_get value
entered_scale	<int>	XView xv_get value
FRAME_CMD_PUSHPIN_IN	<int>	XView xv_get value
XV_KEY_DATA	<Xv_opaque>	XView xv_get value
INSTANCE	<Attr_attribute>	direct access

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Arg#4
Map:: change_size	<int> change_error	<Xv_opaque> popup_frame	<double> new_width	<double> new_length	<int> new_scale

(2) CANONICAL TRACES

$$\text{canonical}(T_i) \leftrightarrow (T_i = _) \vee (T_i = \text{init}(o,p)) \vee (T_i = \text{init}(o,p).\text{show}()) \vee (T_i = \text{init}(o,p).\text{show}().\text{change}(it))$$

$$\vee$$

$$(T_i = \text{init}(o,p).\text{show}().\text{change}(it).\text{change_error})$$

$$\text{canonical}(T_c) \leftrightarrow (T_c = _)$$

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences

- The traces contain exactly the information needed for the equivalences and outputs

AUXILIARY FUNCTIONS

parse(S,S1,S2,S3,S4) =

conditions	equivalences
(S = S1.S2.S3.S4) ∧ (S1 = [init(o,p)] _{i=0} ¹) ∧ (S2 = [show()] _{i=0} ¹) ∧ (S3 = [change(it)] _{i=0} ¹) ∧ (S4 = [change_error] _{i=0} ¹)	true
else	false

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for init, show, change, change_error, and cfChange

T.init(o,p) =

conditions	equivalences
T = _	init(o,p)
T ≠ _	%already_initialized%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- If one LHC condition is true, the other must be false, and they therefore partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- init(o,p) is defined by event itself, other RHC item is error message.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- Only one is specified, init(), and it is canonical.

Consistency (3): All RHC values are unique:

- One is value, one is error.

T.show() =

conditions	equivalences
T = _	%uninitialized%
else	I.show() where parse(T, I, S, C, CE)

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- First case is empty trace, second has something in trace, third is else, insuring partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- First two cases are errors, in last I must be defined since T is not empty.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- Trace init().show() [I.show()] is in the canonical trace.

- Consistency (3): All RHC values are unique:
- True.

$T.change(it) \equiv$

conditions	equivalences
$T = _$	%uninitialized%
$T = init(o,p)$	%undisplayed%
$parse(T, I, S, C, CE) \wedge$ $S \neq _ \wedge I = init(o,p) \wedge$ $p \rightarrow change_size() = TRUE$	equivalence = I.S.change(it); ADD-TO-TRACE(T_p , change_size(f, atof(entered_width), atof(entered_length), entered_scale)) where f is frame created by init()
else	equivalence = I.S.change(it).change_error; ADD-TO-TRACE(T_p , change_size(f, atof(entered_width), atof(entered_length), entered_scale)) where $parse(T, I, S, C, CE) \wedge I = init(o,p) \wedge change_error = change_size()$ $\wedge$ f is frame created by init()

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- First two are obviously different, third has show() in T, else separates third from fourth.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- I and S defined for traces in 3rd/4th cases; p, change_error defined as given; entered* values defined if popup has been created (since init must be in trace, that is true).

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- init().show().change() and init().show().change().change_error are canonical

Consistency (3): All RHC values are unique:

- 3rd + 4th cases differ in equivalence

$T.change_error \equiv$

conditions	equivalences
$T = init(o,p).show().change(it)$	$T.change_error$
else	%undefined%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T defined by LHC.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T.change_error canonical if T is as defined by LHC

Consistency (3): All RHC values are unique:

- One value, one error.

$T_c.cfChange(item,e) \equiv T_c; ADD-TO-TRACE(T_p, change(item))$
where $PopMapSize* p = xv_get(item, XV_KEY_DATA, INSTANCE);$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- If change occurs, the PopMapSize object must have already been created, and p will be valid.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T_c canonical by definition.

- Consistency (3): All RHC values are unique:
- Only one value.

(4) VALUES

OUTPUT VALUES

$V[\text{popup_frame}](T) =$

conditions	values
$T =$	%undefined%
else	frame created via init function

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- frame is defined by init, which must be part of any non-empty trace.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- Either frame or error.

$V[(\text{popup_window})](T) =$

conditions	values
$T =$	%undefined%
$T = \text{init}(o,p)$	%undisplayed%
$T = \text{init}(o,p).\text{show}()$	popup window displayed on screen; Width field = $p \rightarrow \text{width}$ formatted "%.2f"; Length field = $p \rightarrow \text{length}$ formatted "%.2f"; Scale field = $p \rightarrow \text{scale}$; values may be modified by user
$T = \text{init}(o,p).\text{show}().\text{change}(it) \wedge$ $xv_get(\text{frame created by init}(),$ $\text{FRAME_CMD_PUSHPIN_IN}) = \text{TRUE}$	popup fields set to values from $p \rightarrow$ as given above
$T = T1.\text{change}(it) \wedge$ $xv_get(\text{frame created by init}(),$ $\text{FRAME_CMD_PUSHPIN_IN}) = \text{FALSE}$	popup window disappears from screen
else	popup forced to remain on screen, with values as modified by user

Map Size

Width: <f%.2f>
Length: <f%.2f>

Scale: __<int>__
☒ ☒

Change

[PopupMapSize]

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- LHC partitions the entire canonical trace.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- window and fields are created by init, which is included in RHC trace

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- Either has constant (default) appearance or one modified by user input.

RETURN VALUES

(none)

CLASS: WINMAP

TYPE IMPLEMENTED: <WinMap>

(1) SYNTAX

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3
init	<void>	<Xv_opaque>	<Map*>	<PopupLoadMap*>
		#4 <PopupSaveMap*>	#5 <PopupMapSize*>	
unimplemented	<void>			
quit	<void>			
set_title	<void>	<char*> new_title		

OUTPUT VARIABLES

Variable Name	Type	Access
frame	<Xv_opaque>	publicly accessible
(main window)	(XView window)	N/A

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3
cfMenuFileLoad	<Menu_item>	<Menu_item>	<Menu_generate>	
cfMenuFileSave	<Menu_item>	<Menu_item>	<Menu_generate>	
cfMenuFileQuit	<Menu_item>	<Menu_item>	<Menu_generate>	
cfMenuMapRedraw	<Menu_item>	<Menu_item>	<Menu_generate>	
cfMenuMapClear	<Menu_item>	<Menu_item>	<Menu_generate>	
cfMenuMapChangeSize	<Menu_item>	<Menu_item>	<Menu_generate>	
cfMenuUnimplemented	<Menu_item>	<Menu_item>	<Menu_generate>	
cfRepaint		<Canvas>	<Xv_window>	<Display>
		#4 <Window>	#5 <Xv_xrectlist>	
cfDestroy	<Notify_value>	<Xv_opaque>	<Destroy_status>	

CLASS OUTPUT VARIABLES

Variable Name	Type	Access
notify_value	<Notify_value>	func return

INPUT VARIABLES

Variable Name	Type	Access
XV_KEY_DATA	<Xv_opaque>	XView xv_get value
INSTANCE	<Attr_attribute>	direct access

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3
PopupLoadMap::show	<void>			
PopupSaveMap::show	<void>			
PopupMapSize::show	<void>			
Map::init_draw	<void>	<Display> display	<Window> xid	<WinMap*> pWinMap
Map::clear	<void>			

Utils::cfNotice_OK	<void>	<Xv_opaque> owner	<char *> message	
--------------------	--------	----------------------	---------------------	--

(2) CANONICAL TRACES

$\text{canonical}(T_i) \leftrightarrow (T_i = _) \vee (T_i = \text{init}(o, p1, p2)) \vee (T_i = \text{init}(o, p1, p2).set_title(t))$

$\text{canonical}(T_c) \leftrightarrow (T_c = _) \vee (T_c = \text{cfDestroy}(c, s))$

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for access functions init, unimplemented, quit, cfMenuFileQuit, cfMenuMapRedraw, cfMenuMapChangeSize, cfRepaint, and cfDestroy

$T.\text{init}(o, p1, p2, p3, p4) \equiv$

conditions	equivalences
$T = _$	equivalence = $T.\text{init}(o, p1, p2, p3, p4)$; ADD-TO-TRACE(T_{p1} , $\text{init_draw}(\text{disp}, \text{xid})$) where $\text{disp} + \text{xid}$ are defined by XView calls to create the window
else	%already_initialized%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- $\text{init}(o, p)$ is defined by event itself, other RHC item is error message.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- Only one is specified, $\text{init}()$, and it is canonical.

Consistency (3): All RHC values are unique:

- One value, one error.

$T.\text{unimplemented}() \equiv$

conditions	equivalences
$T = _$	%uninitialized%
else	equivalence = T ; ADD-TO-TRACE(T_u , $\text{cfNotice_OK}(f, \text{"This function has not been implemented."})$) where T_u is the class access trace for Utils

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T defined by equivalence.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T is canonical by definition.

Consistency (3): All RHC values are unique:

- One value, one error.

T.quit() =

conditions	equivalences
T = _	%uninitialized%
else	T

Completeness/Consistency same as above.

T.set_title(t) =

conditions	equivalences
T = _	%uninitialized%
T = T1.init(o,p1,p2,p3,p4)	T.set_title(t)
T = T1.set_title(t')	T1.set_title(t)

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- LHC partitions the canonical trace completely.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T and t defined by LHS; T1 defined by LHC

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- init().set_title() canonical; third case LHC and RHC same trace and therefore canonical.

Consistency (3): All RHC values are unique:

- One replaces set_title, one adds.

T_C.cfMenuFileQuit(item, op) =

conditions	equivalences
xv_get(item, XV_KEY_DATA, INSTANCE) = 0	%invalid item%
op = MENU_NOTIFY ^ xv_get(item, XV_KEY_DATA, INSTANCE) ≠ 0	equivalence = T _C ; ADD-TO-TRACE(T _p , quit()) where WinMap* p = xv_get(item, XV_KEY_DATA, INSTANCE);
else	T _C

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first two differentiated by =/≠; else insure partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T_C defined by LHS; if fn called then item must be created and therefore p will be valid.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T_C is canonical by definition.

Consistency (3): All RHC values are unique:

- error + one has ADD-TO-TRACE, other does not.

T_C.cfMenuMapRedraw(item, op) =

conditions	equivalences
xv_get(item, XV_KEY_DATA, INSTANCE) = 0	%invalid item%
op = MENU_NOTIFY ^ xv_get(item, XV_KEY_DATA, INSTANCE) ≠ 0	equivalence = T _C ; ADD-TO-TRACE(T _p , draw()) where Map* p = xv_get(item, XV_KEY_DATA, INSTANCE);
else	T _C

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first two differentiated by $\neq$; else insure partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T_C defined by LHS; if fn called then item must be created and therefore p will be valid.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T_C is canonical by definition.

Consistency (3): All RHC values are unique:

- error + one has ADD-TO-TRACE, other does not.

$T_C.cfMenuMapChangeSize(item, op) \equiv$

conditions	equivalences
$xv_get(item, XV_KEY_DATA, INSTANCE) = 0$	%invalid item%
$op = MENU_NOTIFY \wedge$ $xv_get(item, XV_KEY_DATA, INSTANCE) \neq 0$	equivalence = T_C ; ADD-TO-TRACE(T_p , show()) where $PopupMapSize * p =$ $xv_get(item, XV_KEY_DATA, INSTANCE);$
else	T_C

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first two differentiated by $\neq$; else insure partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T_C defined by LHS; if fn called then item must be created and therefore p will be valid.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T_C is canonical by definition.

Consistency (3): All RHC values are unique:

- error + one has ADD-TO-TRACE, other does not.

$T_C.cfMenuUnimplemented(item, op) \equiv$

conditions	equivalences
$xv_get(item, XV_KEY_DATA, INSTANCE) = 0$	%invalid item%
$op = MENU_NOTIFY \wedge$ $xv_get(item, XV_KEY_DATA, INSTANCE) \neq 0$	equivalence = T_C ; ADD-TO-TRACE(T_p , unimplemented()) where $WinMap * p =$ $xv_get(item, XV_KEY_DATA, INSTANCE);$
else	T_C

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first two differentiated by $\neq$; else insure partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- T_C defined by LHS; if fn called then item must be created and therefore p will be valid.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T_C is canonical by definition.

Consistency (3): All RHC values are unique:

- error + one has ADD-TO-TRACE, other does not.

$T_C.cfRepaint(canvas, pw, display, xid, rects) \equiv$

conditions	equivalences
$xv_get(pw, XV_KEY_DATA, INSTANCE) = 0$	%invalid item%

$op = \text{MENU_NOTIFY} \wedge$ $xv_get(pw, XV_KEY_DATA, INSTANCE) \neq 0$	$equivalence = T_C;$ $ADD\text{-}TO\text{-}TRACE(T_p, draw())$ where Map* p = $xv_get(pw, XV_KEY_DATA, INSTANCE);$
else	T_C

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first two differentiated by $\neq$; else insure partition

Completeness (4): The predicates in the RHIC are defined whenever the corresponding predicate in the LHC is 'true':

- T_C defined by LHS; if fn called then item must be created and therefore p will be valid.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T_C is canonical by definition.

Consistency (3): All RHC values are unique:

- error + one has ADD-TO-TRACE, other does not.

$T_C.cfDestroy(client, status) \equiv cfDestroy(client, status)$

Completeness (3): The predicates in the LHIC of each table partition the intended domain of the relation:

- no partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by LHS

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- $cfDestroy()$ is canonical

Consistency (3): All RHC values are unique:

- one value only

(4) VALUES

OUTPUT VALUES

$V[frame](T) =$

conditions	values
$T =$	%undefined%
$T = \text{init}(o.p1, p2)$	frame id for (main window)

Completeness (3): The predicates in the LHIC of each table partition the intended domain of the relation:

- LHC partitions the canonical trace.

Completeness (4): The predicates in the RHIC are defined whenever the corresponding predicate in the LHC is 'true':

- frame is defined by init.

Consistency (2): All traces specified in the RHIC of the equivalence section are canonical:

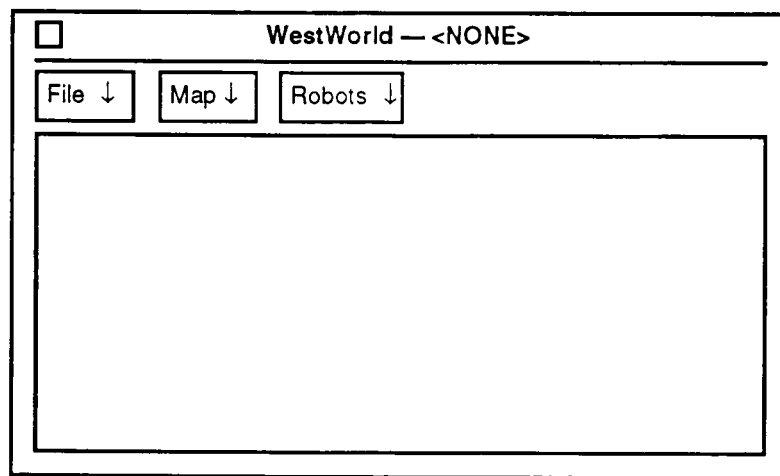
- No traces in RHC.

Consistency (3): All RHC values are unique:

- Either frame or error.

V[(main_window)](T) =

conditions	values
T = _	%undefined%
T = init(o,p1,p2,p3,p4)	display WinMap on screen, with canvas exactly encompassing default map size, with title "WestWorld -- <None>", with border fitting map size (Map::init_draw). with menus as follows: - File: Load..., Save..., Quit - Map: Redraw <default>, <blank>, Update HELIX Map, Clear Map, Change Map Size..., New Map Object... - Robots: Summon... <default>, <blank>, Start All, Stop All, Quit All
T = T1.set_title(t)	same window as above. with title "WestWorld -- t"



Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- LHC partitions the canonical trace.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- window is defined by init.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC.

Consistency (3): All RHC values are unique:

- Window titles differ.

V[notify_value](T_C) =

conditions	values
T _C = _	%undefined%
T _C = cfDestroy(client, status) ∧ status = DESTROY_CLEANUP	notify_next_destroy_func(client, status)
else	NOTIFY_DONE

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- first two differ, else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- constant, error, or client/status defined by LHC

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- Error, constant, or fn call return

RETURN VALUES

Program Name	Argument No	Value
cfDestroy	Value	notify_value

Completeness (2): There is one output function/relation that specifies each output value:

- There is one output value defined above for notify_value.

CLASS: UTILS

TYPE IMPLEMENTED: <Utils>

(1) SYNTAX

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
cfNotice_OK	<void>	<Xv_opaque> owner	<char *> message
cfNotice_YN	<int> yn_answer	<Xv_opaque> owner	<char *> message

CLASS OUTPUT VARIABLES

Variable Name	Type	Access
(notice)	(XView Notice)	N/A
yn_answer	<int>	func return

INPUT VARIABLES

Variable Name	Type	Access
notice_confirm	notice Confirm button	pseudo-event
notice_yn	notice Y/N button value	pseudo-event

(2) CANONICAL TRACES

$$\text{canonical}(T_C) \leftrightarrow (T_C = _) \vee (T_C = \text{cfNotice_OK}(o,m)) \vee (T_C = \text{cfNotice_YN}(o,m)) \vee (T_C = \text{notice_yn})$$

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for cfNotice_OK, cfNotice_YN, notice_confirm, and notice_yn.

$T_C.\text{cfNotice_OK}(o,m) \equiv$

conditions	equivalences
$T_C = \text{cfNotice_OK}(o,m) \vee$ $T_C = \text{cfNotice_YN}(o,m)$	%waiting%
else	cfNotice_OK(o.m)

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RIIC are defined whenever the corresponding predicate in the LHC is 'true':

- event defined by LHC.

Consistency (2): All traces specified in the RIIC of the equivalence section are canonical:

- cfNotice_OK is canonical.

Consistency (3): All RHC values are unique:

- One value, one error.

$T_C.cfNotice_YN(o,m) \equiv$

conditions	equivalences
$T_C = cfNotice_OK(o,m) \vee$ $T_C = cfNotice_YN(o,m)$	%waiting%
else	cfNotice_YN(o,m)

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- event defined by LHC.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- cfNotice_YN is canonical.

Consistency (3): All RHC values are unique:

- One value, one error.

$T_C.notice_confirm \equiv$

conditions	equivalences
$T_C = cfNotice_OK(o,m)$	_
else	%no Confirm notice%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- constants.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- _ is canonical.

Consistency (3): All RHC values are unique:

- One value, one error.

$T_C.notice_yn \equiv$

conditions	equivalences
$T_C = cfNotice_YN(o,m)$	notice_yn
else	%no YN notice%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- notice_yn defined by LHS.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- notice_yn is canonical.

Consistency (3): All RHC values are unique:

- One value, one error.

(4) VALUES

OUTPUT VALUES

$V[(notice)](T) =$

conditions	values
$T = _$	%no_output%
$T = cfNotice_OK(o, m)$	display XView notice with owner o, message m, and Confirm button
$T = cfNotice_YN(o, m)$	display XView notice with owner o, message m, and Yes and No buttons

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Since T is canonical, the conditions partition the canonical trace and therefore give a full partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by LHC

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- One value, one error.

$V[yn_answer](T) =$

conditions	values
$T = notice_yn \wedge$ $notice_yn = NOTICE_YES$	TRUE (1)
$T = notice_yn \wedge$ $notice_yn \neq NOTICE_YES$	FALSE (0)
else	%undefined%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- constants

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- notice_yn is canonical.

Consistency (3): All RHC values are unique:

- Two distinct values, one error.

RETURN VALUES

Program Name	Argument No	Value
cfNotice_YN	Value	yn_answer

Completeness (2): There is one output function/relation that specifies each output value:

- There is one output value defined above for yn_answer.

IV. Clear Boxes

The clear boxes consist of the following files, which are attached:

- ww_ui.H — header file containing class, constant, and misc. definitions
- Main.C — file with main() loop and global variables
- Map.C — class implementation for Map
- PopupLoadSave.C — class implementations for PopupLoadMap + PopupSaveMap
- PopupMapSize.C — class implementation for PopupMapSize
- WinMap.C — class implementation for WinMap
- Utils.C — misc/utility routines

See below for increment 2 C++ headers.

V. Class Design and Class BB Specifications (Second Level)

This section deals with objects or functions that were "discovered" during the implementation of the first level of clear boxes.

(1) Choose candidate objects

During the development of the Map object, it became apparent that there was a need for some additional classes. First, the interaction with files appeared to be similar to dealing with an external object, and therefore it appeared that it would be easier to encase these interactions into a File class, which is defined below. However, it was decided that since File would only be used by Map and would not significantly improve the specification's readability, a separate class was not created. In the code above, filesystem is considered to be an external object with a set of function calls.

In addition, the Map object requires a data structure to hold the Map data. The structure must be able to represent multiple types of objects, specifically boxes and cylinders in this increment and perhaps others in future increments. Therefore, a hierarchy with an abstract base class and subclasses which actually implement the specific object types is appropriate. This results in the MapObject hierarchy, with subclasses MapBox and MapCylinder. This hierarchy was at first included in the first level on this increment, but it was realized that this was inappropriate, since the MapObject hierarchy should only be defined after its requirements are clear from developing the Map class.

(2) Assign top-level stimuli to objects

Not applicable at this level.

(3) Identify inter-class stimuli

MapObject responds to cfSelectAndLoad() by selecting the appropriate MapObject subclass via checking cfIsMe() for each subclass and then creates an instance of the class that has cfIsMe()==TRUE and calls load(Xv_opaque frame, char* line) for the new object. If there are problems with loading, a notice explaining the problem is displayed using the passed frame. MapBox and MapCylinder have to have appropriate class function cfIsMe and instance functions load(), save(), and draw(). The save(char *line, int limit) function takes the current data in the object and creates a loadable file line for the object to be saved to a file via the calling function. The set_next(MapObject*) function sets the next pointer for the object to the given parameter and returns that pointer, next contains the value of the current next pointer. An overloaded next() function was considered to provide both the set_next and next services, but this was deemed unacceptable because of possible confusion during specification and design. MapObject is an abstract superclass, and therefore cannot be instantiated.

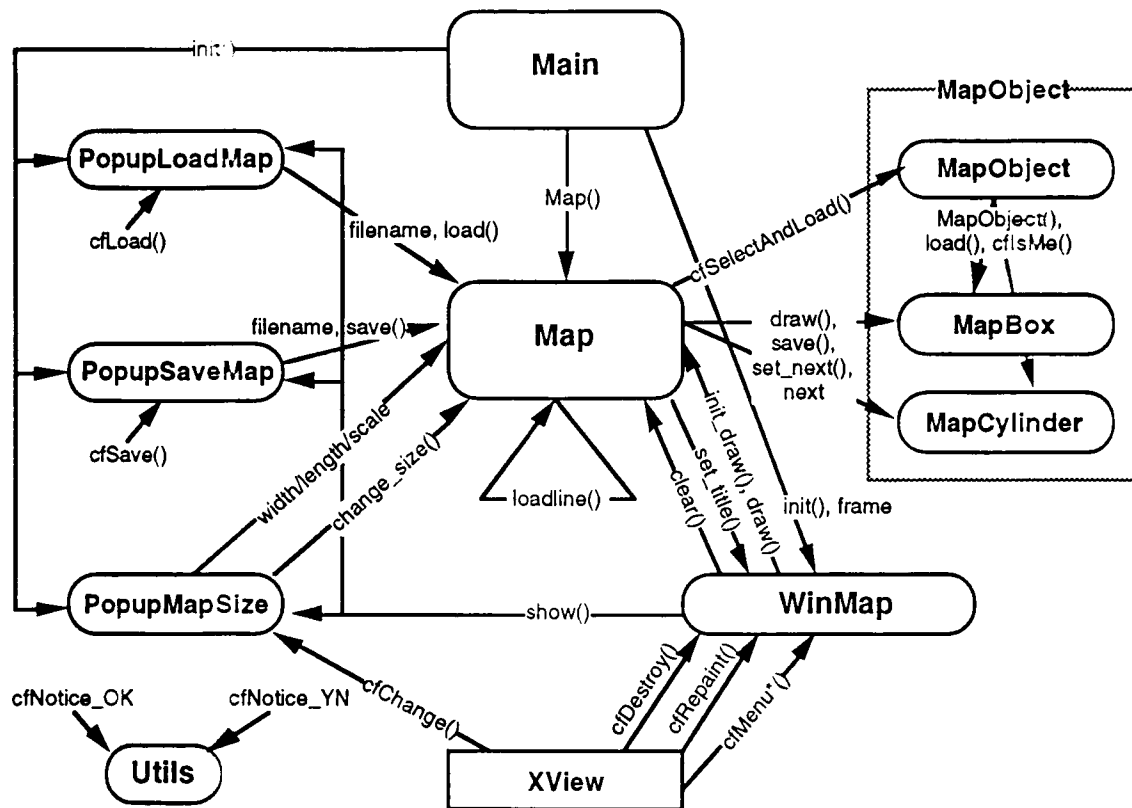


Figure Y: Second Increment Object Interaction Diagram (Final)

(4) Black Box Definitions

define_BB MapObject

access programs

MapObject()

MapObject* set_next(MapObject* nextobj)

virtual int load(Xv_opaque frame, int lineno, char* line)

virtual void save(char *buffer, int bufsize)

virtual void draw(Display *display, Window xid, int scale, int maxy)

output variables

MapObject* next

class access programs

static MapObject* cfSelectAndLoad(Xv_opaque frame, int lineno, char* line)

external access

int MapBox::cfIsMe(char* l)

int MapCylinder::cfIsMe(char* l)

void Utils::cfNotice_OK(char *message)

MapBox* MapBox::new()

MapCylinder* MapCylinder::new()

void MapBox::delete(MapBox*)

void MapCylinder::delete(MapCylinder*)

transition

```

Si = MapObject() --> no response
Si = next --> returns value from last set_next(n) call, otherwise returns NULL
Si = set_next(p) --> p
Si = load(f,n,l) --> not implemented in this class
Si = save(b, bs) --> not implemented in this class
Si = draw(d,xw,s) --> not implemented in this class
Si = cfSelectAndLoad(f,n,l) ∧ MapBox::cfIsMe(l) = TRUE ∧
    (p = new MapBox)->load(f,n,l) = TRUE --> p
Si = cfSelectAndLoad(f,n,l) ∧ MapBox::cfIsMe(l) = TRUE ∧
    (p = new MapBox)->load(f,n,l) = FALSE --> NULL
Si = cfSelectAndLoad(f,n,l) ∧ MapBox::cfIsMe(l) = FALSE ∧
MapCylinder::cfIsMe(l) = TRUE
    (p = new MapCylinder)->load(f,n,l) = TRUE --> p
Si = cfSelectAndLoad(f,n,l) ∧ MapBox::cfIsMe(l) = FALSE ∧
MapCylinder::cfIsMe(l) = TRUE
    (p = new MapCylinder)->load(f,n,l) = FALSE --> NULL
Si = cfSelectAndLoad(f,n,l) ∧ MapBox::cfIsMe(l) = FALSE ∧
MapCylinder::cfIsMe(l) = FALSE -->
    Utils::cfNotice_OK(f, "Map file format error: unknown object @ line <n>");
    return NULL

```

end_BB

define_BB MapBox

access programs

```

MapObject() <inherited>
MapObject* set_next(MapObject*) <inherited>
virtual int load(Xv_opaque frame, int lineno, char* line) <inherited>, <overridden>
virtual void save(char *buffer, int bufsize) <inherited>, <overridden>
virtual void draw(Display *display, Window xid, int scale, int maxy)
    <inherited>, <overridden>

```

output variables

```

MapObject* next <inherited>

```

class access programs

```

static void cfSelectAndLoad(Xv_opaque frame, int lineno, char* line) <inherited>
static int cfIsMe(char* line)

```

external access

```

void Utils::cfNotice_OK(char *message)

```

transition

```

Si = load(f,n,l) ∧ legal_box(l) --> TRUE
Si = load(f,n,l) ∧ not(legal_box(l)) -->
    Utils::cfNotice_OK(f, "Map file format error: bad box definition @ line <n>")
    return FALSE
Si = save(b, bs) -->
    copy information from load() into "box <locx> <locy> <width> <length>
    <height>" with default height if none specified by load() and limited to length
    of bs.
Si = draw(d,xw,s) -->
    draw rectangle at <locx>*s,<locy>*s+<length>*s of size
    <width>*s,<length>*s (origin in bottom LHC of map)
Si = cfIsMe(l) ∧ strcmp(l, "box", 3) = 0 --> TRUE
Si = cfIsMe(l) ∧ strcmp(l, "box", 3) ≠ 0 --> FALSE

```

end_BB

Spec Function

```
[ legal_box(l) ] =  
  [ if l is of form "box <locx> <locy> <width> <length> [<height>]" --> TRUE  
    else --> FALSE ]
```

NOTES:

(1) only new or over-ridden routines are re-defined in a derived class (subclass).

define_BB MapCylinder

access programs

```
MapObject() <inherited>  
MapObject* set_next(MapObject*) <inherited>  
virtual int load(Xv_opaque frame, int lineno, char* line) <inherited>, <overridden>  
virtual void save(char *buffer, int bufsize) <inherited>, <overridden>  
virtual void draw(Display *display, Window xid, int scale, int maxy)  
  <inherited>, <overridden>
```

output variables

```
MapObject* next <inherited>
```

class access programs

```
static void cfSelectAndLoad(Xv_opaque frame, int lineno, char* line) <inherited>  
static int cfIsMe(char* line)
```

external access

```
void Utils::cfNotice_OK(char *message)
```

transition

```
Si = load(f,n,l) ∧ legal_cylinder(l) --> TRUE  
Si = load(f,n,l) ∧ not(legal_cylinder(l)) -->  
  Utils::cfNotice_OK(f,  
    "Map file format error: bad cylinder definition @ line <n>")  
  return FALSE  
Si = save(b, bs) -->  
  copy information from load() into "cylinder <locx> <locy> <radius> <height>"  
  with default height if none specified by load() and limited to length of bs.  
Si = draw(d,xw,s) -->  
  draw circle at <locx>*s,<locy>*s of radius <radius>*s, origin in bottom LHC  
  of map  
Si = cfIsMe(l) ∧ strcmp(l, "cyl", 3) = 0 --> TRUE  
Si = cfIsMe(l) ∧ strcmp(l, "cyl", 3) ≠ 0 --> FALSE
```

end_BB

Spec Function

```
[ legal_cylinder(l) ] =  
  [ if l is of form "cyl[inder] <locx> <locy> <radius> [<height>]" --> TRUE  
    else --> FALSE ]
```

VI. TAM Specifications for Classes (Second Level)

CLASS: MAPOBJECT

TYPE IMPLEMENTED: <MapObject>

(1) SYNTAX

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Other
MapObject	(constructor)				
set_next	<MapObject*>	<MapObject*> nextobj			
load	<int> load_ok	<Xv_opaque> frame	<int> lineno	<char*> line	virtual
save	<void>	<char*> buffer	<int> bufsize		virtual
draw	<void>	<Display*> display	<Window> xid	<int> scale	virtual
		(Arg#4) <int> maxy			

OUTPUT VARIABLES

Variable Name	Type	Access
next	<MapObject*>	public
load_ok	<int>	fn return
buffer	<char*>	fn param return

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3
cfSelectAndLoad	<MapObject*> created	<Xv_opaque> frame	<int> lineno	<char*> line

CLASS OUTPUT VARIABLES

Variable Name	Type	Access
created	<MapObject*>	fn return

INPUT VARIABLES

Variable Name	Type	Access
boxnew	<MapBox*>	ext fn return
cylnew	<MapCylinder*>	ext fn return

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
MapBox::cfIsMe	<int>	<char*> line	
MapCylinder::cfIsMe	<int>	<char*> line	
Utils::cfNotice_OK	<void>	<Xv_opaque> frame	<char*> message
MapBox::new	<MapBox*> boxnew		
MapCylinder::new	<MapCylinder*> cylnew		
MapBox::delete	<void>	<MapBox*>	

MapCylinder::delete	<void>	<MapCylinder*>	
---------------------	--------	----------------	--

(2) CANONICAL TRACES

$\text{canonical}(T_i) \leftrightarrow (T_i = \text{MapObject}()) \vee (T_i = \text{set_next}(n))$

$\text{canonical}(T_c) \leftrightarrow (T_c = _) \vee (T_c = \text{boxnew}) \vee (T_c = \text{cylnew})$

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for MapObject, set_next(), load(), save(), draw(), cfSelectAndLoad(), boxnew, cylnew

$T.\text{MapObject}() \equiv \text{MapObject}()$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- No partitioning of domain, therefore complete

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- MapObject() is a canonical trace

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

$T.\text{set_next}(n) \equiv \text{set_next}(n)$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- set_next() defined by RHS, L & D defined by parsing T.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- set_next() is canonical.

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

$T.\text{load}(f, \text{In}, l) \equiv \% \text{undefined for this class} \%$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- error

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

$T.\text{save}(b, bs) \equiv \% \text{undefined for this class} \%$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- error

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

T.draw(d, xw, s, m) $\equiv$ %undefined for this class%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- error

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- N/A

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

$T_c.cfSelectAndLoad(f, ln, l) \equiv$

conditions	equivalences
MapBox::IsMe(l)=TRUE $\wedge$ (boxnew = new MapBox)->load(f, ln, l)=TRUE	equiv = boxnew; ADD-TO-TRACE(T_{cmb} , IsMe(l)); ADD-TO-TRACE(T_{cmb} , new); ADD-TO-TRACE(T_{boxnew} , load(f, ln, l)), where T_{cmb} is the class trace for MapBox
MapBox::IsMe(l)=TRUE $\wedge$ (boxnew = new MapBox)->load(f, ln, l)=FALSE	equiv = _; ADD-TO-TRACE(T_{cmb} , IsMe(l)); ADD-TO-TRACE(T_{cmb} , new); ADD-TO-TRACE(T_{boxnew} , load(f, ln, l)); ADD-TO-TRACE(T_{cmb} , delete), where T_{cmb} is the class trace for MapBox
MapBox::IsMe(l)=FALSE $\wedge$ MapCyl::IsMe(l)=TRUE $\wedge$ (cylnew = new MapBox)->load(f, ln, l)=TRUE	equiv = cylnew; ADD-TO-TRACE(T_{cmb} , IsMe(l)); ADD-TO-TRACE(T_{cmc} , IsMe(l)); ADD-TO-TRACE(T_{cmc} , new); ADD-TO-TRACE(T_{cylnew} , load(f, ln, l)), where T_{cmb} is the class trace for MapBox and T_{cmc} is the class trace for MapCylinder
MapBox::IsMe(l)=FALSE $\wedge$ MapCyl::IsMe(l)=TRUE $\wedge$ (cylnew = new MapBox)->load(f, ln, l)=FALSE	equiv = _; ADD-TO-TRACE(T_{cmb} , IsMe(l)); ADD-TO-TRACE(T_{cmc} , IsMe(l)); ADD-TO-TRACE(T_{cmc} , new); ADD-TO-TRACE(T_{cylnew} , load(f, ln, l)); ADD-TO-TRACE(T_{cmc} , delete(cylnew)), where T_{cmb} is the class trace for MapBox and T_{cmc} is the class trace for MapCylinder
else	equiv = _; ADD-TO-TRACE(T_{cmb} , IsMe(l)); ADD-TO-TRACE(T_{cmc} , IsMe(l)); ADD-TO-TRACE(T_u , cfNotice_OK(f, "Map file format error: unknown object @ line <n>"), where T_u is the class trace for Utils, T_{cmb} is the class trace for MapBox and T_{cmc} is the class trace for MapCylinder

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- values defined by LHC or LHS

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- _, boxnew, cylnew all canonical for T_c

Consistency (3): All RHC values are unique:

- equiv and ADD-TO-TRACE results all different

$T_c.boxnew \equiv boxnew$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- boxnew defined by LHS

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- boxnew is a canonical trace

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

$T_c.cylnew \equiv cylnew$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- cylnew defined by LHS

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- boxnew is a canonical trace

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

(4) VALUES

OUTPUT VALUES

$V[next](T) =$

conditions	values
$parse(T, I, L, D) \wedge I = set_next(n)$	n
else	NULL

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- n defined in LHC.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- Value or NULL.

$V[load_ok](T) = \%undefined\%$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- no partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- error only.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC.

Consistency (3): All RHC values are unique:

- error only.

$V[buffer](T) = \%undefined\%$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- no partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- error only.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC.

Consistency (3): All RHC values are unique:

- error only.

$V[\text{created}](T_C) =$

conditions	values
$T_C = \text{boxnew}$	value of boxnew
$T_C = \text{cylnew}$	value of cylnew
else	NULL

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined in LHC.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- different values or NULL.

RETURN VALUES

Program Name	Argument No	Value
cfSelectAndLoad	Value	created

Completeness (2): There is one output function/relation that specifies each output value:

- There is one output value $V[\text{created}]$ defined above for the one value in the table.

CLASS: MAPBOX

TYPE IMPLEMENTED: <MapBox>

(1) SYNTAX

Note: (i) items are inherited

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Other
(i) MapObject	(constructor)				
(i) set_next	<MapObject*>	<MapObject*> nextobj			
(i) load	<int> load_ok	<Xv_opaque> frame	<int> lineno	<char*> line	virtual, overridden
(i) save	<void>	<char*> buffer	<int> bufsize		virtual, overridden
(i) draw	<void>	<Display*> display	<Window> xid	<int> scale	virtual, overridden
		(Arg#4) <int> maxy			

OUTPUT VARIABLES

Variable Name	Type	Access
(i) next	<MapObject*>	public
(i) load_ok	<int>	fn return (overridden)
(i) buffer	<char*>	fn param return
(output _screen)	(X display window)	N/A

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3
(i) cfSelectAndLoad	<MapObject*> created	<Xv_opaque> frame	<int> lineno	<char*> line
cfIsMe	<int> isMe	<char *> line		

CLASS OUTPUT VARIABLES

Variable Name	Type	Access
(i) created	<MapObject*>	fn return

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
Utils::cfNotice_OK	<void>	<Xv_opaque> frame	<char*> message

(2) CANONICAL TRACES

canonical(T_i) <--> ($T_i = \text{MapObject}() \vee \text{set_next}(n)$) $\vee$
 ($T_i = [\text{MapObject}() \vee \text{set_next}(n)].\text{load}(f.ln,l)$) $\vee$
 ($T_i = [\text{MapObject}() \vee \text{set_next}(n)].\text{load}(f.ln,l).\text{draw}(d,xw,s,m)$)

$\text{canonical}(T_C) \leftrightarrow (T_C = _) \vee (T_C = \text{boxnew}) \vee (T_C = \text{cylnew}) \vee (T_C = \text{cfIsMe}(l))$

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

AUXILIARY FUNCTIONS

$\text{parse}(S, S1, S2, S3) =$

conditions	equivalences
$(S = S1.S2.S3.S4) \wedge$ $(S1 = [\text{MapObject}() \vee \text{set_next}(n)]) \wedge$ $(S2 = [\text{load}(f, l, l)]_{i=0}^1) \wedge$ $(S3 = [\text{draw}(d, xw, s, m)]_{i=0}^1)$	true
else	false

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

$\text{legal_box}(l) =$

conditions	equivalences
$(l \text{ is of form "box } \langle \text{locx} \rangle \langle \text{locy} \rangle \langle \text{width} \rangle$ $\langle \text{length} \rangle [\langle \text{height} \rangle])"$	true
else	false

Completeness (3): The predicates in the LHIC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for $\text{set_next}()$, $\text{load}()$, $\text{save}()$, $\text{draw}()$, and $\text{cfIsMe}()$; $\text{MapObject}()$, $\text{cfSelectAndLoad}()$, boxnew , cylnew are unchanged from previous

$T.\text{set_next}(n) \equiv \text{set_next}(n).L.D$ where $\text{parse}(T, I, L, D)$

Completeness (3): The predicates in the LHIC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- $\text{set_next}()$ defined by RHIS, L & D defined by parsing T.

Consistency (2): All traces specified in the RHIC of the equivalence section are canonical:

- $\text{set_next}()$, $\text{set_next}().\text{load}()$, and $\text{set_next}().\text{load}().\text{draw}()$ are all canonical traces.

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

$T.load(f, ln, l) \equiv$

conditions	equivalences
legal_box(l)	$I.load(f, ln, l)$ where $parse(T, I, L, D)$
else	$equiv = I.load(f, ln, l)$ where $parse(T, I, L, D)$; ADD-TO-TRACE(T_u , cfNotice_OK(f, "Map file format error: bad box definition @ line <ln>") where T_u is the class trace for Utils

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by LHS

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- $I.load()$ is canonical

Consistency (3): All RHC values are unique:

- trace alone or trace + ADD-TO-TRACE().

$T.save(b, bs) \equiv T$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- no partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by LHS

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T is canonical by definition

Consistency (3): All RHC values are unique:

- only one

$T.draw(d, xw, s, m) \equiv$

conditions	equivalences
$parse(T, I, L, D) \wedge L = load(f, ln, l) \wedge legal_box(l)$	$I.L.draw(d, xw, s, m)$
else	%cannot draw without legal load() first%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by LHS

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- $I.load()$ is canonical

Consistency (3): All RHC values are unique:

- trace alone or trace + ADD-TO-TRACE().

$T_C.cfIsMe(l) \equiv cfIsMe(l)$

Completeness (3): The predicates in the LIIC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- $cfIsMe()$ defined by LHS

Consistency (2): All traces specified in the RIIC of the equivalence section are canonical:

- $cfIsMe()$ is a canonical trace

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

(4) VALUES

OUTPUT VALUES

Note: V[next] and V[created] are unchanged from inherited; V[load_ok] and V[buffer] override superclass def.

V[load_ok](T) =

conditions	values
parse(T, I, L, D) ∧ L=load(f,ln,l) ∧ legal_box(l)	TRUE (1)
parse(T, I, L, D) ∧ L=load(f,ln,l) ∧ not(legal_box(l))	FALSE (0)
else	%undefined%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- constants or error.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC.

Consistency (3): All RHC values are unique:

- opposite values or error.

V[buffer](T) =

conditions	values
parse(T, I, L, D) ∧ L=load(f,ln,l) ∧ legal_box(l)	"box <locx> <locy> <width> <length> <height>" from load() with default height if none specified
else	""

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHIC are defined whenever the corresponding predicate in the LHC is 'true':

- value or error.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC.

Consistency (3): All RHC values are unique:

- value or error.

V[(output_screen)](T) =

conditions	values
parse(T, I, L, D) ∧ L=load(f,ln,l) ∧ legal_box(l) ∧ D=draw(d,xw,s,m)	draw rectagle parsed from l in window defined by d, xw with scale s and positioned relative to bottom LHC of window
else	%undefined%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by RHC.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- one value or error.

$V[isMe](T_C) =$

conditions	values
$T_C = cfIsMe(l) \wedge$ $strncmp("box", l, 3) = 0$	TRUE (1)
$T_C = cfIsMe(l) \wedge$ $strncmp("box", l, 3) \neq 0$	FALSE (0)
else	%undefined%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- constants.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- opposite values or error.

RETURN VALUES

Program Name	Argument No	Value
(i) load	Value	load_ok (overridden)
(i) save	Arg#1	buffer (overridden)
(i) cfSelectAndLoad	Value	created
cfIsMe	Value	isMe

Completeness (2): There is one output function/relation that specifies each output value:

- Yes, except for inherited values that are not overridden.

CLASS: MAPCYLINDER

TYPE IMPLEMENTED: <MapCylinder>

(1) SYNTAX

Note: (i) items are inherited

ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3	Other
(i) MapObject	(constructor)				
(i) set_next	<MapObject*>	<MapObject*> nextobj			
(i) load	<int> load_ok	<Xv_opaque> frame	<int> lineno	<char*> line	virtual, overridden
(i) save	<void>	<char*> buffer	<int> bufsize		virtual, overridden
(i) draw	<void>	<Display*> display	<Window> xwin	<int> scale	virtual, overridden

OUTPUT VARIABLES

Variable Name	Type	Access
(i) next	<MapObject*>	public
(i) load_ok	<int>	fn return (overridden)
(i) buffer	<char*>	fn param return
(output _screen)	(X display window)	N/A

CLASS ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2	Arg#3
(i) cfSelectAndLoad	<MapObject*> created	<Xv_opaque> frame	<int> lineno	<char*> line
cfIsMe	<int> isMe	<char *> line		

CLASS OUTPUT VARIABLES

Variable Name	Type	Access
(i) created	<MapObject*>	fn return

EXTERNAL ACCESS PROGRAMS

Func Name	Value	Arg#1	Arg#2
Utils::cfNotice_OK	<void>	<Xv_opaque> frame	<char*> message

(2) CANONICAL TRACES

canonical(T_i) $\leftrightarrow$ ($T_i = \text{MapObject}() \vee \text{set_next}(n)$) $\vee$
 $(T_i = [\text{MapObject}() \vee \text{set_next}(n)].\text{load}(f, \text{ln}, l)) \vee$
 $(T_i = [\text{MapObject}() \vee \text{set_next}(n)].\text{load}(f, \text{ln}, l).\text{draw}(d, xw, s, m))$

 canonical(T_c) $\leftrightarrow$ ($T_c = _$) $\vee$ ($T_c = \text{boxnew}$) $\vee$ ($T_c = \text{cylnew}$) $\vee$ ($T_c = \text{cfIsMe}(l)$)

Consistency (1): The canonical form fulfills the requirements of section XI.

- The traces in the set are not further reducible when passed through the equivalences
- The traces contain exactly the information needed for the equivalences and outputs

AUXILIARY FUNCTIONS

parse(S,S1,S2,S3) =

conditions	equivalences
(S = S1.S2.S3.S4) $\wedge$ (S1 = [MapObject() $\vee$ set_next(n)]) $\wedge$ (S2 = [load(f,ln,l)] _{i=0} ¹) $\wedge$ (S3 = [draw(d,xw,s,m)] _{i=0} ¹)	true
else	false

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- Else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

legal_cylinder(l) =

conditions	equivalences
(l is of form "cyl[index] <locx> <locy> <radius> [<u><height></u>]")	true
else	false

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- Constants, therefore always defined.

Consistency (3): All RHC values are unique:

- True.

(3) EQUIVALENCES

Completeness (1): There is one equivalence for each event class.

- There is one each for set_next(), load(), save(), draw(), and cflsMe(): MapObject(), cfSelectAndLoad(), boxnew, cylnew are unchanged from previous

T.set_next(n) $\equiv$ set_next(n).L.D where parse(T, I, L, D)

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- set_next() defined by RHS, L & D defined by parsing T.

Consistency (2): All traces specified in the RHIC of the equivalence section are canonical:

- set_next(), set_next().load(), and set_next().load().draw() are all canonical traces.

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

$T.load(f, ln, l) \equiv$

conditions	equivalences
$legal_cylinder(l)$	$I.load(f, ln, l) \text{ where } parse(T, I, L, D)$
else	$equiv = I.load(f, ln, l) \text{ where } parse(T, I, L, D);$ ADD-TO-TRACE(T_U , cfNotice_OK(f, "Map file format error: bad cylinder definition @ line <ln>"), where T_U is the class trace for Utils

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by LHS

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- $I.load()$ is canonical

Consistency (3): All RHC values are unique:

- trace alone or trace + ADD-TO-TRACE().

$T.save(b, bs) \equiv T$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- no partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by LHS

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- T is canonical by definition

Consistency (3): All RHC values are unique:

- only one

$T.draw(d, xw, s, m) \equiv$

conditions	equivalences
$parse(T, I, L, D) \wedge L = load(f, ln, l) \wedge legal_cylinder(l)$	$I.L.draw(d, xw, s, m)$
else	%cannot draw without legal load() first%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partition

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by LHS

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- $I.load()$ is canonical

Consistency (3): All RHC values are unique:

- trace alone or trace + ADD-TO-TRACE().

$T_C.cflsMe(l) \equiv cflsMe(l)$

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- No partitioning of domain, therefore complete

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- $cflsMe()$ defined by LHS

Consistency (2): All traces specified in the RHIC of the equivalence section are canonical:

- cflsMe() is a canonical trace

Consistency (3): All RHC values are unique:

- No partitioning, therefore unique.

(4) VALUES

OUTPUT VALUES

Note: V[next] and V[created] are unchanged from inherited; V[load_ok] and V[buffer] override superclass def.

V[load_ok](T) =

conditions	values
parse(T, I, L, D) ∧ L=load(f,ln,l) ∧ legal_cylinder(l)	TRUE (1)
parse(T, I, L, D) ∧ L=load(f,ln,l) ∧ not(legal_cylinder(l))	FALSE (0)
else	%undefined%

Completeness (3): The predicates in the LHIC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- constants or error.

Consistency (2): All traces specified in the RHIC of the equivalence section are canonical:

- No traces in RHC.

Consistency (3): All RHC values are unique:

- opposite values or error.

V[buffer](T) =

conditions	values
parse(T, I, L, D) ∧ L=load(f,ln,l) ∧ legal_cylinder(l)	"cylinder <locx> <locy> <radius> <height>" from load() with default height if none specified
else	""

Completeness (3): The predicates in the LHIC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- value or error.

Consistency (2): All traces specified in the RHIC of the equivalence section are canonical:

- No traces in RHC.

Consistency (3): All RHC values are unique:

- value or error.

$V[(output_screen)_i](T) =$

conditions	values
$parse(T, I, L, D) \wedge$ $L = load(f, ln, l) \wedge legal_cylinder(l)$ $\wedge D = draw(d, xw, s, m)$	draw circle parsed from l in window defined by d, xw with scale s and positioned relative to bottom LHC of window
else	%undefined%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- defined by RHC.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- one value or error.

$V[isMe](T_C) =$

conditions	values
$T_C = cfIsMe(l) \wedge$ $strncmp("cyl", l, 3) = 0$	TRUE (1)
$T_C = cfIsMe(l) \wedge$ $strncmp("cyl", l, 3) \neq 0$	FALSE (0)
else	%undefined%

Completeness (3): The predicates in the LHC of each table partition the intended domain of the relation:

- else insures partitioning.

Completeness (4): The predicates in the RHC are defined whenever the corresponding predicate in the LHC is 'true':

- constants.

Consistency (2): All traces specified in the RHC of the equivalence section are canonical:

- No traces in RHC

Consistency (3): All RHC values are unique:

- opposite values or error.

RETURN VALUES

Program Name	Argument No	Value
(i) load	Value	load_ok (overridden)
(i) save	Arg#1	buffer (overridden)
(i) cfSelectAndLoad	Value	created
cfIsMe	Value	isMe

Completeness (2): There is one output function/relation that specifies each output value:

- Yes, except for inherited values that are not overridden.

VII. Clear Boxes (Second Level)

The clear boxes consist of the following files, which are attached:

MapObject.C — class implementation of MapObject, MapBox, and MapCylinder

Increment 2 C++ Header Definitions (Complete)

```
ww_ui.H

WestWorld

Alex L. Bangs, 2/10/93
-----
Modification History:
    2/10/93 ALB Increment 1
    6/21/93 ALB Increment 2

#ifndef WW_UI_HEADER
#define WW_UI_HEADER

#include <math.h>

    Map constants

const double default_width = 12.0;
const double default_length = 12.0;
const int default_scale = 40;
const int min_scale = 1;
const int max_scale = 100;
const double min_width = 1.0;
const double min_length = 1.0;
const int panel_text_size = 80;
const int filename_size = 80;
const double default_obj_height = 2.0;

    simple #define functions

#define min(a,b) ((a) < (b) ? (a) : (b))
#define scaleIt(coord) (rint((coord) * scale))

    Main descriptor
    note no real class for Main, but has function + globals
    class Main
        void main(int argc, char **argv);
extern Attr_attribute    INSTANCE;

extern class MapObject;
extern class WinMap;

    Other class descriptors
class Map
    Display          *display;
    Window           xid;
    GC               gc;

    MapObject*       objects;
    WinMap*          pWinMap;

public:
    double width, length;
```

```

    int scale;
    char filename[filename_size];

    Map();
    ~Map();
    void    init_draw(Display*, Window, WinMap*);
    void    draw();
    void    clear();
    int     change_size(Xv_opaque frame,
                        double new_width, double new_length, int new_scale);
    int     load(Xv_opaque frame, char *loadfile);
    int     save(Xv_opaque frame, char *savefile);
    int     loadline(Xv_opaque frame, int lineno, char *line);
};

class PopupLoadMap {
    Xv_opaque    frame;
    Xv_opaque    controls;
    Xv_opaque    filename_field;
    Xv_opaque    button;

    Map*    pMap;
    void    update();

public:
    void    init(Xv_opaque owner, Map* pTheMap);
    void    show();
    void    load(Panel_item item);

// class functions
    static void    cfLoad(Panel_item item, Event *event);
};

class PopupSaveMap {
    Xv_opaque    frame;
    Xv_opaque    controls;
    Xv_opaque    filename_field;
    Xv_opaque    button;

    Map*    pMap;
    void    update();

public:
    void    init(Xv_opaque owner, Map* pTheMap);
    void    show();
    void    save(Panel_item item);

// class functions
    static void    cfSave(Panel_item item, Event *event);
};

class PopupMapSize {
    Xv_opaque    frame;
    Xv_opaque    controls;
    Xv_opaque    map_width_field;
    Xv_opaque    map_length_field;
    Xv_opaque    map_scale_field;
    Xv_opaque    change_button;

    Map*    pMap;
    void    update();           // update numbers in the window
};

```

```

public:
    void      init(Xv_opaque owner, Map* pTheMap);
    void      show();           // redisplay the box, and do an update
    void      change(Panel_item); // change button pressed; send values to
pMap

// class functions
    static void cfChange(Panel_item item, Event *event);
        // XView button callback for Change
};

class WinMap {
    Xv_opaque      controls;
    Xv_opaque      file_menu_button;
    Xv_opaque      map_menu_button;
    Xv_opaque      robots_menu_button;
    Xv_opaque      canvas;
    Xv_window      canvas_paint;
    Display* display;
    Window         xid;

    Xv_opaque      file_menu_create(caddr_t *, Xv_opaque);
    Xv_opaque      map_menu_create(caddr_t *, Xv_opaque);
    Xv_opaque      robots_menu_create(caddr_t *, Xv_opaque);

    Map*           pMap;
    PopupLoadMap*  pPopupLoadMap;
    PopupSaveMap*  pPopupSaveMap;
    PopupMapSize*  pPopupMapSize;

public:
    Xv_opaque      frame;

    void      init(Xv_opaque owner, Map*, PopupLoadMap*, PopupSaveMap*,
        PopupMapSize*);
    void      unimplemented();
    void      quit();
    void      set_title(char* new_title);

    // XView interface callbacks (class functions)
    static Menu_item cfMenuFileLoad(Menu_item item, Menu_generate op);
    static Menu_item cfMenuFileSave(Menu_item item, Menu_generate op);
    static Menu_item cfMenuFileQuit(Menu_item item, Menu_generate op);
    static Menu_item cfMenuMapRedraw(Menu_item item, Menu_generate op);
    static Menu_item cfMenuMapClear(Menu_item item, Menu_generate op);
    static Menu_item cfMenuMapChangeSize(Menu_item item, Menu_generate op);
    static Menu_item cfMenuUnimplemented(Menu_item item, Menu_generate op);

    // general XView callbacks (class functions)
    static Notify_value cfDestroy(Xv_opaque client, Destroy_status status);
    static void cfRepaint(Canvas canvas, Xv_window paint_window,
        Display *display, Window xid, Xv_xrectlist *rects);
};

class Utils {
public:
    // class functions
    static void cfNotice_OK(Xv_opaque owner, char* message);
    static int  cfNotice_YN(Xv_opaque owner, char* message);
};

class MapObject {

```

```

protected:
    double locx, locy, height;

public:
    MapObject*      next;

    MapObject();
    MapObject*      set_next(MapObject* nextobj);
    virtual int      load(Xv_opaque frame, int lineno, char* line) = 0;
    virtual void      save(char* buffer, int bufsize) = 0;
    virtual void      draw(Display* display, Window xid, int scale, int maxy) =
0;

// class functions
    static MapObject* cfSelectAndLoad(Xv_opaque frame, int lineno,
                                     char* line);
};

class MapBox : public MapObject {
// private
    double width, length;

public:
    int      load(Xv_opaque frame, int lineno, char* line);
    void      save(char* buffer, int bufsize);
    void      draw(Display* display, Window xid, int scale, int maxy);

// class functions
    static int cfIsMe(char* line);
};

class MapCylinder : public MapObject {
// private
    double radius;

public:
    int      load(Xv_opaque frame, int lineno, char* line);
    void      save(char* buffer, int bufsize);
    void      draw(Display* display, Window xid, int scale, int maxy);

// class functions
    static int cfIsMe(char* line);
};

#endif

```

Vita

Alex L. Bangs was born in Midland, Michigan on July 23, 1966. He grew up in Michigan and moved to Indiana, where he started his first professional programming job at the age of 14. He attended Harvard University, where he was active in the International Relations Council and worked in the Harvard Robotics Laboratory. In 1988 he received an A.B. degree in Computer Science and Engineering Sciences *magna cum laude*.

After graduation, he worked for a year at the Institute for Defense Analyses in Alexandria, Virginia as a Research Staff Member where he concentrated on technology policy analysis. He next worked at Honeybee Robotics in New York City as a Project Engineer, developing space and commercial robotic prototypes including a robot bartender. In 1990, he moved to Tennessee to work at Oak Ridge National Laboratory. The same year he began work on his Master of Science degree at the University of Tennessee, concentrating in software engineering, and worked during the 1991-1992 school year as a research assistant. He graduated in August 1993.

Since 1990, he has been a Research Associate in the Intelligent Systems Section at Oak Ridge National Laboratory, most recently concentrating in cooperating mobile robots research. He has also been an ongoing computing consultant to Bangs Laboratories of Carmel, Indiana since its incorporation in 1988.

The author is a member of ACM and IEEE.